
Acorn Assembler

Copyright © 1994 Acorn Computers Limited. All rights reserved.

Some updates and changes copyright © 2002 Castle Technology Ltd. All rights reserved.

Some updates and changes copyright © 2011 RISC OS Open Ltd. All rights reserved.

Issue 1 published by Acorn Computers Technical Publications Department.

Issue 2 published by Castle Technology Ltd.

Issue 3 published by RISC OS Open Ltd.

No part of this publication may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or stored in any retrieval system of any nature, without the written permission of the copyright holder and the publisher, application for which shall be made to the publisher.

The product described in this manual is not intended for use as a critical component in life support devices or any system in which failure could be expected to result in personal injury.

The product described in this manual is subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by the publisher in good faith. However, the publisher cannot accept any liability for any loss or damage arising from the use of any information or particulars in this manual.

All trademarks are the property of their respective owners.

Published by RISC OS Open Ltd.

Issue 1, December 1994 (Acorn part number 0484,233).

Issue 2, October 2002 (updates by Castle Technology Ltd).

Issue 3, July 2011 (updates by RISC OS Open Ltd).

Contents

Contents iii

Introduction 1

- Assembler tools 2
- This user guide 2
- Conventions used in this manual 4

Part 1 – Using the assembler 7

ObjAsm 9

- Starting ObjAsm 9
- The SetUp dialogue box 11
- The SetUp menu 14
- ObjAsm output 27
- ObjAsm icon bar menu 28
- Example ObjAsm session 29
- ObjAsm command lines 30

Part 2 – Assembly language details 39

ARM assembly language 41

- General 41
- Input lines 41
- AREAs 41
- ORG and ABS 43
- Instruction sets and syntax 44
- Object file format 44
- Symbols 44
- Labels 45
- Local labels 46
- Comments 47
- Constants 47
- The END directive 47

CPU instruction set 49

- Extended range immediate constants 49

- The MOV32 instruction 50
- The ADR instruction 50
- The ADRL instruction 51
- The IT instruction 51
- The UND instruction 52
- Literals 52
- Shifts by zero 53

Floating point instructions 55

- Floating point constants 56
- Extended range immediate constants 58
- The VMOV2 instruction 58
- Register comparison instructions 59
- 2 x 32-bit vector zip and unzip 59
- Literals 60
- Right shifts by zero 61
- Fixed point conversions with zero fractional bits 61
- Unsigned saturation of signed numbers 61

Directives 63

- Storage reservation and initialisation – DCB, DCW, DCD etc 63
- Binary file inclusion – BIN and INCBIN 65
- Floating point store initialisation – DCFH, DCFS and DCFD 65
- Describing the layout of store – MAP and FIELD 66
- Organisational directives – END, ORG, LTORG, KEEP and LEAF 66
- Links to other object files – IMPORT, EXPORT, etc 67
- Links to other source files – GET/INCLUDE 69
- Diagnostic generation – ASSERT, ! and INFO 70
- Dynamic listing options – OPT 71
- Titles – TTL and SUBT 71
- Miscellaneous directives – ALIGN, NOFP, RLIST and ENTRY 72

Symbolic capabilities 73

- Setting constants 73
- Local and global variables – GBL, LCL and SET 74
- Variable substitution – \$ 75
- Aliases 75
- Built-in variables 76

Expressions and operators 81

- Unary operators 81
- Binary operators 83

Conditional and repetitive assembly 87

- Conditional assembly 87
- Repetitive assembly 90

Macros 91

- Syntax 92
- Local variables 93
- MEXIT directive 94
- Default values 94
- Macro substitution method 95
- Nesting macros 95
- A division macro 96

Part 3 – Developing software for RISC OS 99

PSR Manipulation 101

Writing relocatable modules in assembler 103

- Assembler directives 104
- Example 105

Interworking assembler with C 107

- Examples 107

Part 4 – Appendices 111

Changes to the assembler 113

Differences from RVDS 119

Error messages 123

Example assembler fragments 139

- Using the conditional instructions 139
- Pseudo-random binary sequence generator 140
- Multiplication by a constant 141
- Loading a word from an unknown alignment 142
- Sign/zero extension of a half word 142
- Setting condition codes 142
- Full multiply 144

Support for AAsm source 145

- The --absolute option 145

Index 147

1

Introduction

Acorn Assembler is a development environment for producing RISC OS desktop applications and relocatable modules written in ARM assembly language. It consists of a number of programming tools which are RISC OS desktop applications. These tools interact in ways designed to help your productivity, forming an extendable environment integrated by the RISC OS desktop. Acorn Assembler may be used with Acorn C/C++ (a part of this product) to provide an environment for mixed C, C++ and assembler development.

This product includes tools to:

- edit program source and other text files
- search and examine text files
- examine some binary files
- assemble small assembly language programs
- assemble and construct more complex programs under the control of makefiles, these being set up from a simple desktop interface
- squeeze finished program images to occupy less disk space
- construct linkable libraries
- debug RISC OS desktop applications interactively
- design RISC OS desktop interfaces and test their functionality
- use the Toolbox to interact with those interfaces.

Most of the tools in this product are also of general use for constructing applications in other programming languages, such as C and C++. These non-language-specific tools are described in the accompanying *Desktop Tools* guide.

Installation

Installation of Acorn Assembler is described in the chapter *Installing Acorn C/C++* on page 7 of the accompanying *Desktop Tools* guide.

Assembler tools

The assembler provided includes the following features:

- full support of the ARM instruction set, for all versions up to and including architecture 7, as used in the Cortex series of processors
- support for ARM THUMB, up to and including THUMB version 3
- global and local label capability
- powerful macro processing
- comprehensive expression handling
- conditional assembly
- repetitive assembly
- comprehensive symbol table printouts
- pseudo-opcodes to control printout.

ObjAsm

The Assembler ObjAsm creates object files which cannot be executed directly, but must first be linked using the Link tool. It is often most efficient to construct larger programs from several portions, assembling each portion with ObjAsm before linking them all together with Link. Object files linked with those produced by ObjAsm may be produced from some programming language other than assembler, for example C.

The Link tool is described in the chapter *Link* on page 137 of the accompanying *Desktop Tools* guide.

This user guide

This document is a reference guide to ObjAsm, which is the only tool in this product which is not used for programming in other languages. The others are described in the accompanying *Acorn C/C++* and *Desktop Tools* guides. It is assumed that you are familiar with other relevant Archimedes documentation, such as the:

- *Welcome Guide* supplied with your computer
- *RISC OS 3 User Guide*
- *RISC OS 3 Programmer's Reference Manual*.

Recommended Books

One or more of these books will be useful if you are writing a lot of ARM assembler.

- *ARM Architecture Reference Manual*, Second Edition, edited by David Seal : Addison-Wesley, 2000, 816 pages, ISBN 0-201-73719-1.
This book is also known as the 'ARM ARM' and is an essential reference for anyone working at a low level with the ARM processor, but its style makes it unsuitable as introductory reference.
The paper version has not been re-issued since architecture 5TE, but the electronic version receives updates a few times per year. It is available in PDF format free of charge from ARM's website, although you do need to register first. The manual had a major reworking after architecture 6, when the 'UAL' assembler syntax was introduced, and it is now distributed in separate editions for profile 'M' CPUs and other CPUs. The final pre-UAL version of the manual is also still available. See:
ARMv7-AR Architecture Reference Manual
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html>
ARMv7-M Architecture Reference Manual
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403c/index.html>
ARMv5 Architecture Reference Manual (includes pre-UAL architecture 6)
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0100i/index.html>
- *ARM 7500FE Data Sheet*, document number ARM DDI 0077B, ARM Ltd, 1996, 365 pages, and *CL-PS7500FE Advance Data Book*, document number 447500-001, Cirrus Logic, 1997, 251 pages. These include the official documentation of the final hardware implementation of the FPA, so represent the best and most easily-obtained reference for that part of the instruction set. Note that these instructions are not included in the ARM ARM.
- *Intel XScale Core Developer's Manual*, Intel Corporation, 2004, 220 pages. The definitive reference for the XScale coprocessor 0.
- *ARM Assembly Language: Fundamentals and Techniques*, William Hohl, CRC Press, 2009, 371 pages, ISBN 978-1439806104.
Despite the publication date, this reportedly only describes architectures up to 4T.
- *ARM Assembly Language - an Introduction*, J.R. Gibson : Lulu Enterprises, 2007, 244 pages, ISBN 978-1847536969.
- *ARM System-on-Chip Architecture*, 2nd Edition, Prof. Steve Furber : Addison-Wesley, 2000, 432 pages, ISBN 978-0201675191.
By one of the original designers of the ARM, this is now showing its age, and only covers up to architecture 5TE.

- *The ARM RISC Chip – A Programmer’s Guide*, A. van Someren and C. Atack – Wokingham, UK: Addison-Wesley, 1993, 400 pages, ISBN 0201624109. This is a good introduction to the ARM although the book is now rather dated and only covers up to architecture 3.
- *Archimedes Assembly Language: A Dabhand Guide*, second edition, M. Ginns – Manchester, UK: Dabs Press, 1988, 368 pages, ISBN 1870336208. Out of print and difficult to obtain, but useful as it specifically refers to using RISC OS and the built-in BBC BASIC assembler.
- *ARM Assembly Language Programming*, P.J. Cockerell – Computer Concepts/MTC, 1987, ISBN 0951257900. Out of print and difficult to obtain. Only covers architecture 2a, of historic interest.

Note on program examples

Both general and specific examples of syntax and screen output are given, but there are occasions where the full syntax of an instruction and its accompanying screen appearance would obscure the specific points being made. It follows, therefore, that not all the examples given in the text can be used directly since they are incomplete.

Conventions used in this manual

The Assembler has its own interpretations of the punctuation symbols and special symbols which are available from the keyboard. These are:

!	“	#	\$	%	&	^	@	()
[]	{	}		:	.	,	;	+
-	/	*	=	<	>	?	_		

In order to distinguish between characters used in syntax and descriptive or explanatory characters, typewriter style typeface is used to indicate both text which appears on the screen and text which can be typed on the keyboard. This is so that the position of relevant spaces is clearly indicated.

The following typographical conventions are used throughout this manual:

Convention	Meaning
<i>filename</i>	Text that you must replace with the name of a file, register, variable or whatever is indicated.
&1C	Hexadecimal numbers are preceded with an ampersand.

Convention	Meaning
«<i>instruction</i>»	Italic guillemots «» enclose optional items in the syntax. For example, the Assembler ObjAsm accepts a three field source line which may be expressed in the form: «<i>instruction</i>» «<i>label</i>» «<i>;comment</i>»
ALIGN	Text that you type exactly as it appears in the manual. For example: L321 ADD Ra,Ra,Ra,LSL #1 ;multiply by 3

Part 1 – Using the assembler





ObjAsm is the ARM assembler forming part of the Acorn C/C++ product. It processes text files containing program source written in ARM assembly language into linkable object files. Object files can be linked by the Link tool with each other or with libraries of object files to form executable image files or relocatable modules. ObjAsm multitasks under the RISC OS desktop, allowing other tasks to proceed while it operates.

The sources for large programs can be split into several files, each of which only need be re-assembled to an object file when you have altered it.

An example use of ObjAsm would be to construct a binary image file **!RunImage** in a RISC OS desktop application from the two source files **s.interface** and **s.portable**. ObjAsm processes the source files to form **o.interface** and **o.portable**, which the Link tool processes to form **!RunImage**.

The controls of ObjAsm are similar to those of other non-interactive Desktop tools, with the common features described in the chapter *General features* on page 99 of the accompanying *Desktop Tools* guide. You adjust options for the next assembly operation on a **SetUp** dialogue box and menu which by default appear when you click Select on the main icon or drag a source file to it. Once you have set options you click on a **Run** action icon and the assembly starts. While the assembly is running output windows display any text messages from the assembler and allow you to stop the job if you wish.

There is no file type to double click on to start ObjAsm – it owns no file type unlike, for example, Draw.

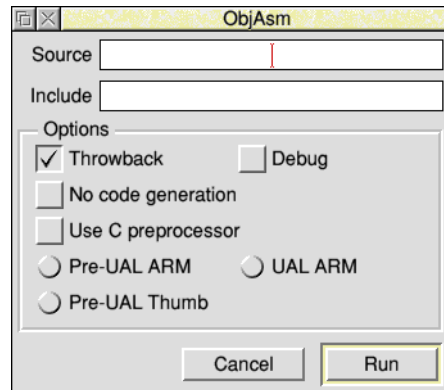
Starting ObjAsm

Like other non-interactive Desktop tools, ObjAsm can be used under the management of Make, with its assembly options specified by the *makefile* passed to Make. For such managed use, ObjAsm is started automatically by Make; you don't have to load ObjAsm onto the icon bar.

To use ObjAsm directly, unmanaged by Make, first open a directory display on the **AcornC_C++.Tools** directory, then double click Select on !ObjAsm. The ObjAsm main icon appears on the icon bar:



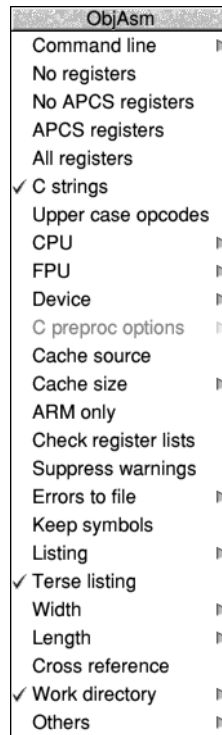
Clicking Select on this icon or dragging an assembly language source file from a directory display to this icon brings up the ObjAsm **Setup** dialogue box:



Source will appear containing the name of the last filename entered there, or empty if there isn't one.

Dragging a file on to the icon will bring up the dialogue box and automatically insert the dragged filename as the **Source** file.

Clicking Menu on the SetUp dialogue box brings up the ObjAsm **SetUp** menu:



The SetUp dialogue box and menu specify the next assembly job to be done. You start the next job by clicking **Run** on the dialogue box (or Command line menu dialogue box). Clicking **Cancel** removes the SetUp dialogue box and clears any changes you have just made to the options settings back to the state before you brought up the SetUp box. The options last until you adjust them again or !ObjAsm is reloaded. You can also save them for future use with an option from the main icon menu.

The SetUp dialogue box

When the SetUp dialogue box is displayed the **Source** writable icon contains the name of the source file to be assembled. The sourcefile can be specified in two ways:

- If the SetUp box is obtained by clicking on the main ObjAsm icon, it comes up with the source file from the previous setting. This helps you repeat a previous assembly, as clicking on the **Run** action icon repeats the last job if there was one.

- If the Setup box appears as a result of dragging a source file containing assembly language text to the main icon, the source file will be the same as the dragged source file.

When the Setup box appears the Source icon has input focus, and can be edited in the normal RISC OS fashion. If a further source file is selected in a directory display and dragged to **Source**, its name replaces the one already there.

Include

The **Include** Setup dialogue box icon adds directories to the source file search path so that arguments to GET/INCLUDE directives (see page 146), or #include directives if you are using the C preprocessor, do not need to be fully qualified. The search rule used is similar to the ANSI C search rule – the current place being the directory in which the current file was found.

The directories are searched in the order in which they are given in the **Include** icon.

Options

The **Throwback** option switches editor throwback on (the default) or off. When enabled, if the DDEUtils module and SrcEdit (or another throwback-enabled editor such as StrongEd or Zap) are loaded, any assembly errors cause the editor to display an error browser. Double clicking Select on an error line in this browser makes the editor display the source file containing the error, with the offending line highlighted. For more details, see the chapter *SrcEdit* on page 69 of the accompanying *Desktop Tools* guide.

The **Debug** option switches on or off the production of debugging tables. When enabled, extra information is included in the output object file which enables source level debugging of the linked image (as long as Link's **Debug** option is also enabled) by the DDT debugger. If this option is disabled, any image file finally produced can only be debugged at machine level. Source level debugging allows the current execution position to be indicated as a displayed line of your source, whereas machine level debugging only shows the position on a disassembly of memory.

The **No code generation** option switches off pass 2 of the assembly, so no output file is generated. This allows you to check the syntax of source code or directives - the most significant omission will be range-checking of immediate constants, since this is only done in pass 2.

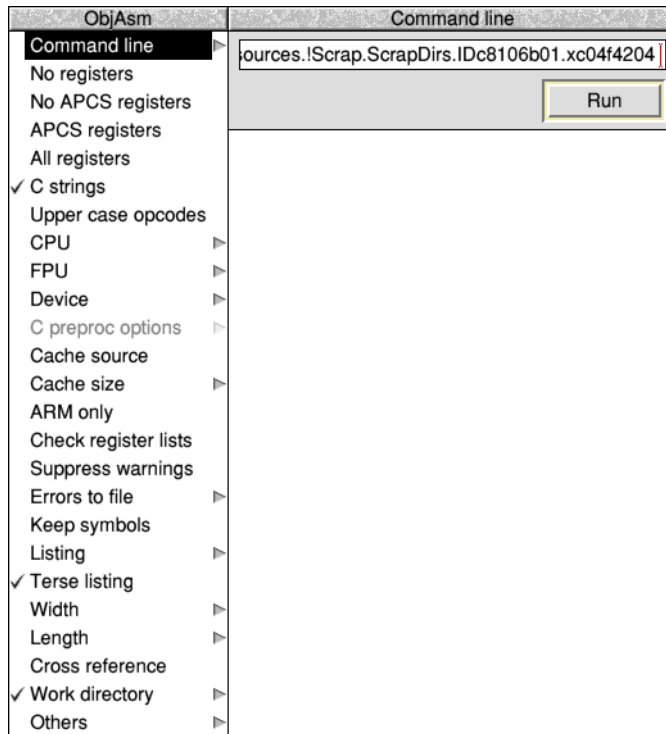
The **Use C preprocessor** option switches on a preprocessing pass over the source code using the C compiler. This permits you to use C statements like `#define`, and most usefully permits the sharing of header files between C and assembler. ObjAsm understands a range of C operators, which permits simple expressions to be included in these header files.

The **Pre-UAL ARM**, **UAL ARM** and **Pre-UAL Thumb** options determine the default instruction set and syntax with which the source code is interpreted, in the absence of a `CODE32`, `ARM` or `CODE16` directive (see page 44). `ARM` means ARM's traditional fixed-width 32-bit instruction set, and `Thumb` means ARM's mixed-width 16-bit / 32-bit instruction set. `UAL` was a significant revision of the instruction syntax which was introduced alongside architecture ARMv6T2, when Thumb's capabilities reached a par with the ARM instruction set, and it became desirable for it to be possible for the same source code to be assembled both as ARM and Thumb. ObjAsm does not yet support `UAL Thumb` assembly. If none of these options are selected, ObjAsm currently behaves as though **Pre-UAL ARM** had been chosen.

The Setup menu

The command line

The ObjAsm RISC OS desktop interface works by driving an ObjAsm tool underneath with a command line constructed from your Setup options. The **Command line** item at the top of the Setup menu leads to a small dialogue box in which the command line equivalent of the current Setup options is displayed:



The **Run** action icon in this dialogue box starts assembly in the same way as that in the main Setup box. Pressing Return in the writable icon in this box has the same effect. Before starting assembly from the command line box, you can edit the command line textually, although this is not normally useful.

Controlling syntax

The next few entries in the Setup menu all control the acceptable syntax for the Assembler.

The first four entries are mutually exclusive options for which register names are pre-declared by ObjAsm. If none are selected, ObjAsm behaves as though **APCS registers** was chosen.

No registers specifies that no register names are pre-declared at all.

No APCS registers specifies that the following register (and coprocessor) names are pre-declared:

- **R0-R15** and **r0-r15**
- **SP, sp, LR, lr, PC** and **pc**
- **acc0-acc7**
- **c0-c15**
- **D0-D31** and **d0-d31**
- **F0-F7** and **f0-f7**
- **p0-p15**
- **Q0-Q15** and **q0-q15**
- **S0-S31** and **s0-s31**

APCS Registers additionally pre-declares register names defined by the ARM Procedure Call Standard: **a1-a4, v1-v5, fp** and **ip**, as well as some registers from the list **v6-v8, SB, sb, SL, sl, FP** and **IP**, depending upon the options passed to **--apcs**.

All registers pre-declares all of the above, irrespective of the **--apcs** options.

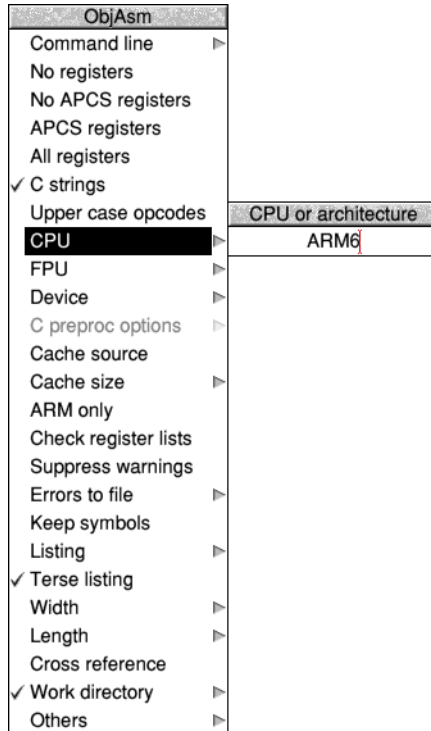
You can specify other APCS variants using the **--apcs** option in the **Others** writable field at the bottom of the menu; see *Specifying other command line options* on page 18, and *Command line options not available from the desktop* on page 23.

C strings, when enabled, allows the assembler to accept C style string escapes such as `'\n'`. **C strings** is enabled by default.

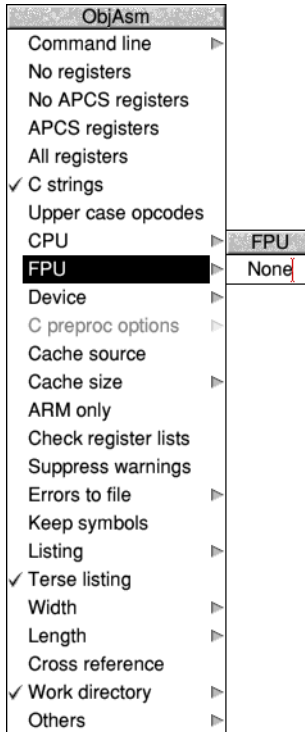
Upper case opcodes, when chosen, makes ObjAsm recognise instruction mnemonics only if they are entirely in upper case. By default, **Upper case opcodes** is not chosen, and ObjAsm recognises mnemonics that are entirely in upper or lower case (but not a mixture of both).

This option is provided mainly to support old code that might have used lower case versions of instruction mnemonics as macro names; it allows the macros to still be recognised as such.

CPU sets the target ARM core or architecture version. To obtain a list of values this can take, enter **list** here and run ObjAsm (no assembly will take place). If nothing is specified here, and **Device** is not specified either, ObjAsm currently defaults to a generic architecture 3 CPU. Some processor specific instructions will produce warnings if assembled for the wrong ARM core.



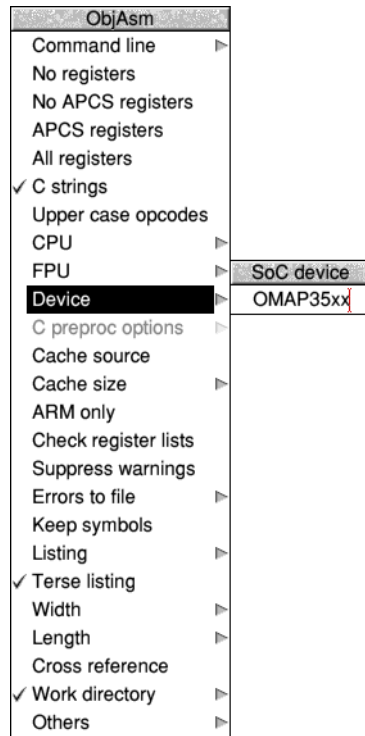
FPU sets the target floating-point unit, and optionally the APCS attributes which flag the floating point calling standard variant in use.



To obtain a list of values this can take, enter **list** here and run ObjAsm (no assembly will take place). Values containing the string **SoftFPA** specify a calling standard which is compatible with software floating point libraries (floating point arguments and results are passed in integer registers and/or the stack) and double-precision parameters are stored using FPA endianness rules. **SoftVFP** is similar, but indicates that double-precision parameters are stored using VFP endianness rules.

The remaining part of the string determines which instructions are warned about if you assemble them for the wrong floating point unit. If nothing is specified here, ObjAsm will first attempt to choose the FPU which naturally accompanies the specified **CPU** or **Device**; if neither was specified, or if only an architecture was specified, the FPU defaults to **FPE2** unless the architecture only supports the Thumb instruction set, in which case the FPU defaults to **None**.

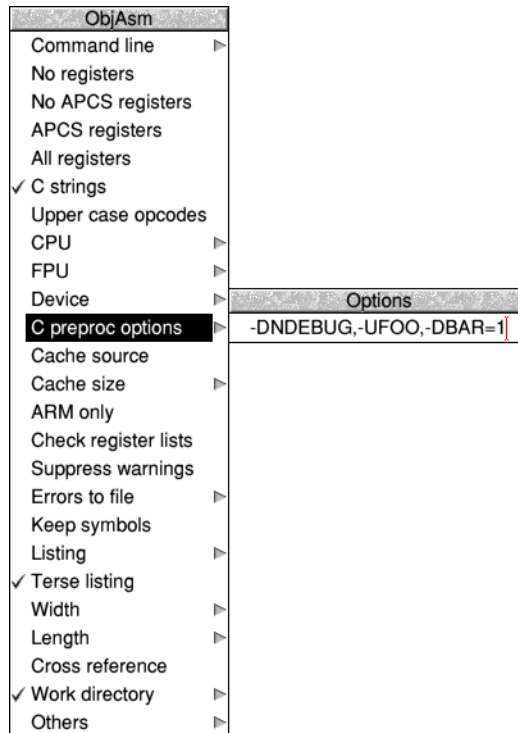
Device sets both the CPU and the FPU, given the name of a system-on-chip.



To obtain a list of values this can take, enter **list** here and run ObjAsm (no assembly will take place).

C preprocessor options

The **C preproc options** entry is available if **Use C preprocessor** is enabled in the SetUp dialogue box. This permits you to specify additional command-line options to pass to the C compiler when it is invoked by ObjAsm – mostly useful for predefining preprocessor variables.



Any include path specified for ObjAsm is automatically passed on to the C compiler, so there is no need to specify it again here. If you need to specify more than one option (which would normally be achieved with a space separator on the C command line) you must substitute a comma character.

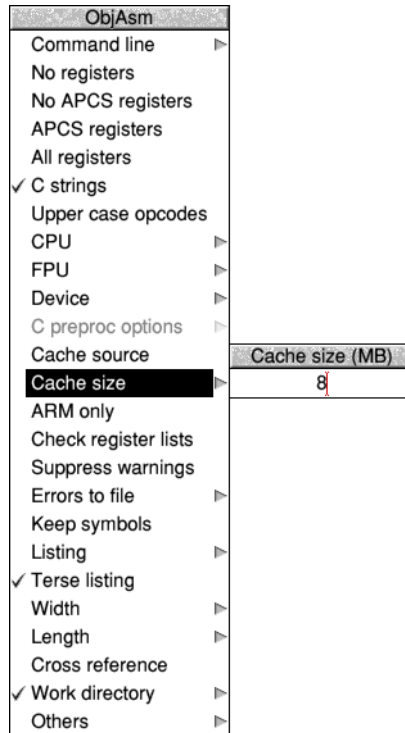
Controlling cacheing

ObjAsm is a two pass assembler – it examines each source file twice. To avoid reading each source file twice from disk the assembler can cache the source in memory, reading it from disk for the first pass, then storing it in RAM for the second. This makes very heavy use of memory, and so is unsuitable for smaller machines.

The next two menu options control this caching:

Cache source enables caching when chosen. By default, caching is disabled.

Cache size allows you to specify the maximum amount of RAM to be used for caching source files, provided that **Cache source** is on. The maximum cache is specified in megabytes; the default is 8MB:



Handling warnings and errors

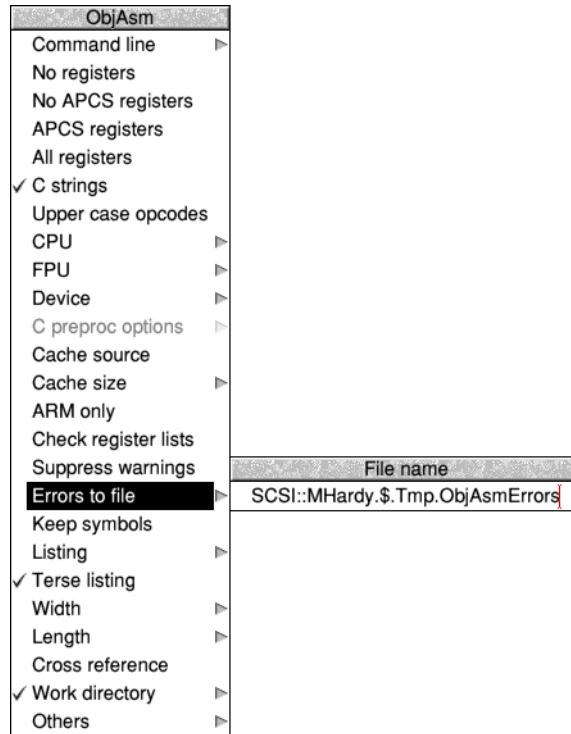
The next menu options control handling of warnings and errors:

ARM only will generate an error if the source file attempts to use Thumb code.

Check register lists will generate a warning if registers in lists in FLDM, FSTM, LDM, POP, PUSH, STM, VLD n , VLDM, VPOP, VPUSH, VST n , VSTM, VTBL or VTBX instructions or RLIST directives are not specified in increasing numeric order.

Suppress warnings, when chosen, turns off the warning messages that ObjAsm generates. It is off by default (i.e. warning messages are generated).

Errors to file allows you to specify a file to which error messages are output for later inspection:



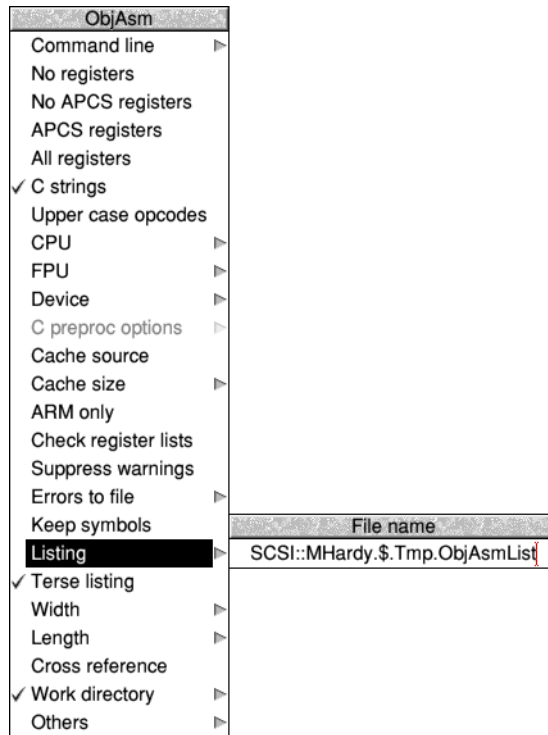
Output options

Keep symbols forces all label definitions to be retained in the output object file. This is equivalent in functionality to the KEEP directive.

Listings

The next options control whether or not a listing is produced, and its format:

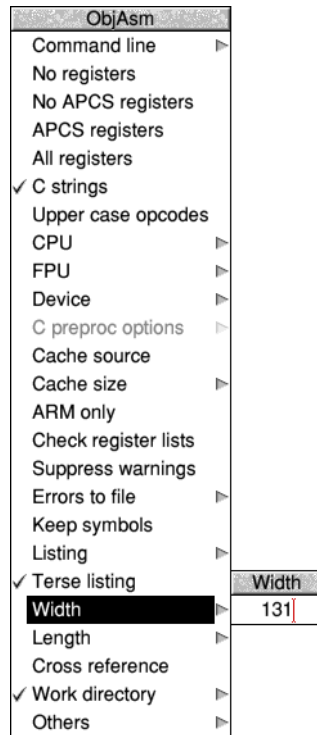
The **Listing** option enables assembler source code to be sent to a file:



This option turns on the Assembler listing, and during assembly the source code, object code, memory addresses and reference line numbers will be sent to the named file. **Listing** is off by default.

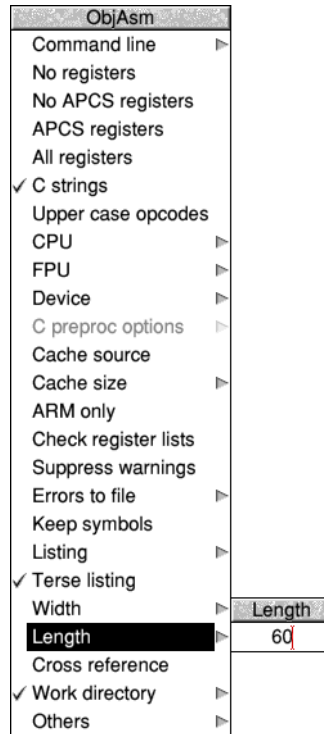
Terse listing modifies the listing that is output such that conditionally non-assembled parts of your program are omitted. **Terse listing** is on by default.

Width sets the width, in characters, of the listing that is output:



This should be between 1 and 254. The default width is 131; a width of 76 is suitable for a Mode 12 RISC OS window.

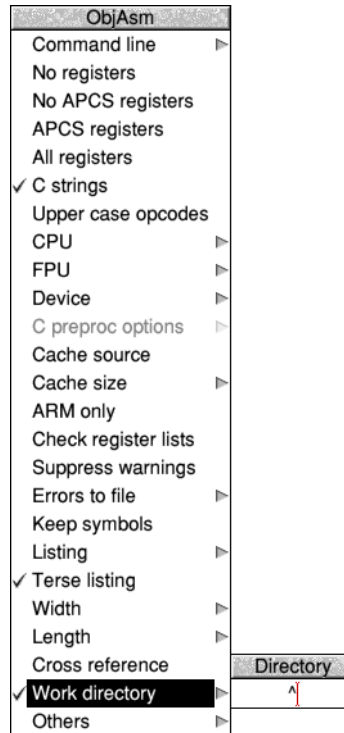
Length sets the number of lines per page for printer output. At the end of each page ObjAsm inserts a form feed character. The default length is 60:



If you choose **Cross reference**, then after assembly ObjAsm outputs an alphabetically sorted cross reference of all symbols encountered. Note that the text output may be very large for a big program, and so this option may not function on a machine with restricted memory. **Cross reference** is off by default.

Choosing your work directory

Work directory allows you to specify the work directory:



The GET and LNK directives both result in the assembler loading source files specified with the directive. The work directory is the place where these source files are to be found. An example is a source file:

```
adfs::HardDisc4.$ .Source.s.foo
```

containing the line:

```
GET s.macros
```

If the work directory is ^ then the file loaded is:

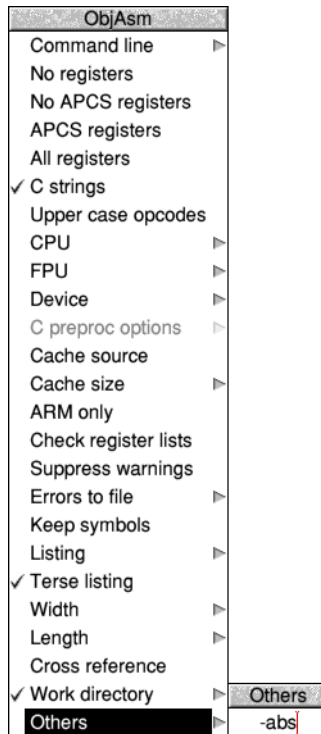
```
adfs::HardDisc4.$ .Source.s.^ .s.macros
(i.e. adfs::HardDisc4.$ .Source.s.macros)
```

The work directory must be given relative to the position of the source file containing the GET or LNK, without a trailing dot.

The default work directory is ^.

Specifying other command line options

The **Others** option on the Setup menu leads to a writable icon in which you can add an arbitrary extra section of text to the command line to be passed to ObjAsm:



This facility is useful if you wish to use any feature which is not supported by any of the other entries on the Setup dialogue box and menu. This may be because the feature is used very little, or because it may not be supported in the future.

For a full description of command line options, see *ObjAsm command lines* on page 22.

ObjAsm output

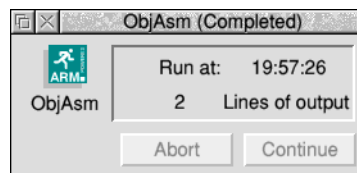
ObjAsm outputs text messages as it proceeds. These include source listings and symbol cross references (as described in the previous sections). By default any such text is directed into a scrollable output window:



This window is read-only; you can scroll up and down to view progress, but you cannot edit the text without first saving it. To indicate this, clicking Select on the scrollable part of this window has no effect.

The contents of the window illustrated above are typical of those you see from a successful assembly; the title line of the assembler with version number, followed by no error messages.

Clicking Adjust on the close icon of the output window switches to the output summary dialogue box. This presents a reminder of the tool running (ObjAsm), the status of the task (Running, Paused, Completed or Aborted), the time when the task was started, and the number of lines of output that have been generated (ie those that are displayed by the output window):



Clicking Adjust on the close icon of the summary box returns to the output window.

Both the above ObjAsm output displays follow the standard pattern of those of all the non-interactive Desktop tools. The common features of the non-interactive Desktop tools are covered in more detail in the chapter *General features* on page 99 of the accompanying *Desktop Tools* guide. Both ObjAsm output displays and the

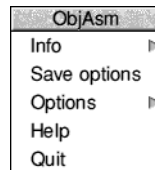
menus brought up by clicking Menu on them offer the standard features, which allow you to abort, pause or continue execution (if the execution hasn't completed), to save output text to a file, or to repeat execution.

ObjAsm error messages appear in the output viewer, with copies in the editor error browser when throwback is working. The appendix *Error messages* on page 183 of this manual contains a list of common ObjAsm error messages together with brief explanations.

Assembly listings and cross references appearing in the output window are often very large for assemblies of complex source files. The scrolling of the output window is useful to view them. To investigate them with the full facilities of the source editor, you can save the output text straight into the editor by dragging the output file icon to the SrcEdit main icon on the icon bar.

ObjAsm icon bar menu

The ObjAsm main icon bar menu follows the standard pattern for non-interactive Desktop tools:

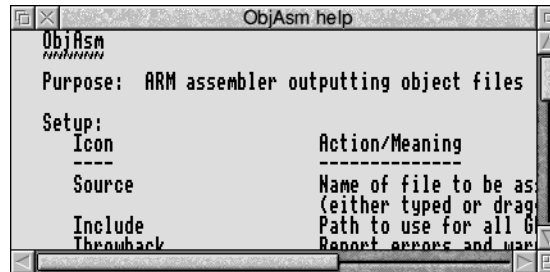


Save options saves all the current ObjAsm options, including both those set from the SetUp dialogue box and from the **Options** item on this menu. When ObjAsm is restarted it is initialised with these options rather than the defaults.

The **Options** submenu allows you to set the following options:

- **Display** specifies the output display as either a text window (default) or as a summary box.
- If **Auto run** is enabled, dragging a source file to the ObjAsm main icon immediately starts an assembly with the current options rather than displaying the SetUp box first. **Auto run** is off by default.
- If **Auto save** is enabled output image files are saved to suitable places automatically without producing a save dialogue box for you to drag the file from. **Auto save** is off by default.

Clicking on **Help** on the main ObjAsm menu displays a short text summary of the various Setup options, in a scrollable read-only window:

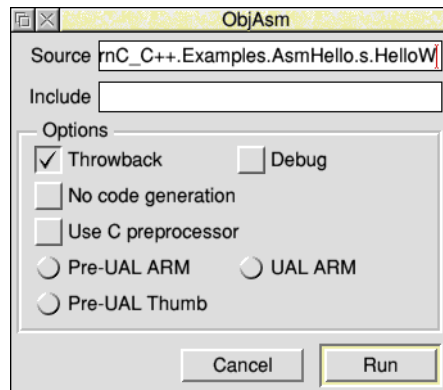


Example ObjAsm session

The programming example **AcornC_C++.Examples.AsmHello** is a non-desktop free standing command line program written in assembly language. It outputs the text 'Hello World'.

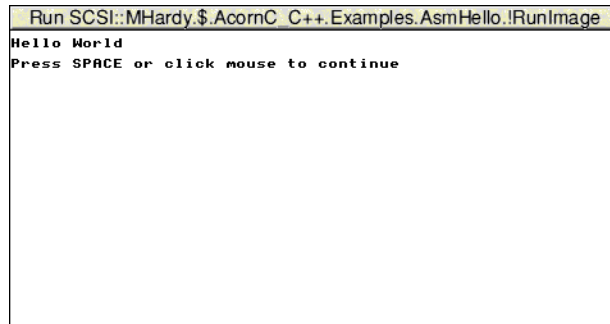
The assembly language source is held in the **s** subdirectory, in the file **HelloW**. The code demonstrates the ObjAsm directives needed for a free standing program;

To assemble **HelloW**, first run !Objasm and !Link by double clicking on them. Drag the **HelloW** source text file to the ObjAsm icon. The **Setup** dialogue box of ObjAsm appears. Check that the default **Setup** options are enabled:



Click on **Run** to proceed, and save the object file produced in the **o** subdirectory. Drag the object file to the Link icon, and **Run** Link to produce an AIF executable image file, the link having the **HelloW** object file as its only input file. Save the image file in **AcornC_C++.Examples.AsmHello.!RunImage**. The command line program is now ready for use.

To run the program under the desktop, double click on it. A window appears with the text 'Hello World':



As the window instructs you to do, press the space bar or click on your mouse. The window disappears.

ObjAsm command lines

ObjAsm, in common with the other non-interactive Desktop tools, can be driven with a text command line without its RISC OS desktop interface appearing. This enables ObjAsm to be driven by Make as specified in textual makefiles.

You can use ObjAsm outside the RISC OS desktop from its command line, in the same way that it could be used in the previous Acorn Desktop Assembler product. However, as all the useful ObjAsm features can be more conveniently used from the RISC OS desktop there is little reason for you to do this. The desktop removes the need for you to understand the command line syntax.

The ObjAsm RISC OS desktop interface drives the ObjAsm tool underneath by issuing a command line constructed from your SetUp options. The Command line SetUp menu option allows you to view the command line constructed in this way.

The Make tool allows you to construct makefiles with assembly operations specified using the ObjAsm desktop interface (by following the Tool options item of Make). You can therefore construct makefiles without understanding the command line syntax of ObjAsm.

The command to invoke ObjAsm takes either of the forms:

```
ObjAsm <options> sourcefile objectfile  
ObjAsm <options> -o objectfile sourcefile
```

The options are listed below, split into two sections: those for which there is a direct equivalent in the SetUp dialogue box or menu, and those others for which there is no equivalent. Where a long-form option takes a parameter separated from

the option by an = character, in the short form, this should be separated by a space character instead. Long-form options differed in syntax in previous releases of ObjAsm; these are not shown below, but are still supported for compatibility's sake. Note that to understand what many of these options do it may be necessary to refer to some of the documentation above.

Command line options available from the desktop

The table below shows the various command line options that correspond to the options available from the SetUp dialogue box and menu, together with a reference to the desktop equivalent, which you should see for full details of the option.

Command line option	Short form	Desktop equivalent	Page
<code>-i dir«<i>,dir</i>»</code>		Include writable icon in dialogue box	12
<code>--throwback</code>	<code>-tb</code>	Throwback option icon in dialogue box	12
<code>--debug</code>	<code>-g</code>	Debug option icon in dialogue box	12
<code>--no_code_gen</code>		No code generation option icon in dialogue box	12
<code>--cpreproc</code>		Use C preprocessor option icon in dialogue box	13
<code>--32</code>	<code>-32</code>	Pre-UAL ARM radio icon in dialogue box	13
<code>--arm</code>		UAL ARM radio icon in dialogue box	13
<code>--16</code>	<code>-16</code>	Pre-UAL Thumb radio icon in dialogue box	13
<code>--regnames=none</code>		No registers in menu	15
<code>--regnames=callstd</code> plus <code>--apcs=none</code>		No APCS registers in menu	15
<code>--regnames=callstd</code>		APCS registers in menu	15
<code>--regnames=all</code>		All registers in menu	15
<code>--no_esc</code>	<code>-noe</code>	C strings in menu	15
<code>--uppercase</code>	<code>-u</code>	Upper case opcodes in menu	15
<code>--cpu=ARMcore</code>	<code>-cpu</code>	CPU in menu	16
<code>--fpu=FPU</code>		FPU in menu	17
<code>--device=device</code>		Device in menu	18
<code>--cpreproc_opts=options</code>		C preproc options in menu	19

Command line option	Short form	Desktop equivalent	Page
<code>--no_cache</code>	<code>-noc</code>	Cache source in menu	20
<code>--maxcache=n</code>	<code>-mc</code>	Cache size in menu	20
<code>--arm_only</code>		ARM only in menu	20
<code>--checkreglist</code>		Check register lists in menu	20
<code>--no_warn</code>	<code>-now</code>	Suppress warnings in menu	20
<code>--errors=errorfile</code>	<code>-e</code>	Errors to file in menu	21
<code>--keep</code>		Keep symbols in menu	21
<code>--list<=listingfile></code>			
	<code>-list</code>	Listing in menu	22
<code>--no_terse</code>	<code>-not</code>	Terse listing in menu	22
<code>--width=n</code>	<code>-wi</code>	Width in menu	23
<code>--length=n</code>	<code>-l</code>	Length in menu	24
<code>--xref</code>	<code>-x</code>	Cross reference in menu	24
<code>--desktop=dirname</code>	<code>-dt</code>	Work directory in menu	25

Command line options not available from the desktop

The table below shows those command line options for which there is no direct equivalent in the SetUp dialogue box or menu. Should you need to use any of these more esoteric options from the desktop, you can add them to the SetUp menu's **Others** option (see *Specifying other command line options* on page 18).

Command line option	Short form	Description
<code>--help</code>	<code>-h</code>	Outputs a summary of the command line options.
<code>--via=filename</code>	<code>-via</code>	Reads in extra command line arguments from the given <i>filename</i> .
<code>--littleend</code>	<code>-li</code>	Assemble code suitable for a little-endian ARM. Sets the built-in variable <code>{ENDIAN}</code> to "little".
<code>--bigend</code>	<code>-bi</code>	Assemble code suitable for a big-endian ARM. Sets the built-in variable <code>{ENDIAN}</code> to "big".

Command line option	Short form	Description
<code>--apcs=«option»«/qualifier»«/qualifier...»</code>		
	<code>-apcs</code>	<p>Specifies whether the ARM Procedure Call Standard is in use, and also specifies some attributes of AREAs. There are three APCS options: none, 3 and an empty string. By default, the register names R0-R15, r0-r15, SP, sp, LR, lr, PC and pc are pre-declared. Unless you use option none, the following register names are also pre-declared: a1-a4, v1-v5, fp, and ip. If you use option 3, then depending on the /reentrant, /swstackcheck and /fp qualifiers, some of sb, s1 and v6-v8 are predeclared as well. If you use an empty option string, then all the above are predeclared, along with upper-case names SB, SL, FP and IP.</p> <p>The default behaviour is to use the 3/noropi/norwpi/32bit/swstackcheck/fp/nointerwork/fpa/fpe2/hardfp/nofpregargs APCS variant used by RISC OS.</p> <p>The qualifiers – which cannot be used with option none – are as follows:</p>
	<code>/pic</code> or <code>/ropi</code>	Sets the PIC attribute for any code AREAs and sets the built-in variable {ROPI} to {TRUE} .
	<code>/nopic</code> or <code>/noropi</code>	Does not affect the PIC attribute for any code AREAs. Sets the built-in variable {ROPI} to {FALSE} . This is the default setting.
	<code>/reentrant</code> or <code>/reent</code> or <code>/pid</code> or <code>/rwpi</code>	Sets the reentrant attribute for any code AREAs, sets the built-in variables {REENTRANT} and {RWPI} to {TRUE} , and if option is 3 , predeclares sb (static base) in place of v6 . Any imported read-write data symbols are taken to be relative to sb .
	<code>/nonreentrant</code> or <code>/nonreent</code> or <code>/nopid</code> or <code>/norwpi</code>	Does not affect the reentrant attribute for any code AREAs. Sets the built-in variables {REENTRANT} and {RWPI} to {FALSE} , and if option is 3 , predeclares v6 in place of sb . This is the default setting.

Command line option	Short form	Description
<code>/32bit</code> or <code>/32</code>		Is the default setting and informs the Linker that the code being generated is written for both 26 and 32 bit ARMs. The built-in variable <code>{CONFIG}</code> is also set to <code>32</code> .
<code>/26bit</code> or <code>/26</code>		Tells the Linker that the code is only intended for 26 bit ARMs. The built-in variable <code>{CONFIG}</code> is also set to <code>26</code> . Note that these options do not of themselves generate particular ARM-specific code, but allow the Linker to warn of any mismatch between files being linked, and also allow programs to use the standard built-in variable <code>{CONFIG}</code> to determine what code to produce.
<code>/swstackcheck</code> or <code>/swst</code>		Does not affect the no-stack-check attribute for any code AREAs. If option is <code>3</code> , predeclares <code>s1</code> . This is the default setting.
<code>/noswstackcheck</code> or <code>/nosw</code>		Sets the no-stack-check attribute for any code AREAs. If option is <code>3</code> , predeclares an additional v-register, <code>v6</code> if reentrant, <code>v7</code> if not.
<code>/fp</code>		The default setting, used when function entry and exit use the <code>fp</code> register as a stack frame pointer.
<code>/nofp</code>		For use when you are not using a stack frame pointer. This cannot be indicated in object files, so the only extent to which this is enforced is that ObjAsm warns you if you attempt to use the <code>FP</code> or <code>fp</code> register names. If option is <code>3</code> , predeclares <code>v8</code> .

Command line option	Short form	Description
<code>/interwork</code> or <code>/inter</code>		Sets the ARM/Thumb interworking attribute for any code AREAs. The built-in variable <code>{INTER}</code> is also set to <code>{TRUE}</code> .
<code>/nointerwork</code> or <code>/nointer</code>		Does not affect the ARM/Thumb interworking attribute for any code AREAs. The built-in variable <code>{INTER}</code> is also set to <code>{FALSE}</code> . This is the default setting.
<code>/fpa</code>		This is the default setting if an FPA FPU is selected. Does not affect the VFP attribute for any code or data AREAs. Sets FPA endianness rules for double-precision floating point numbers (most significant word is always stored in lower register number or lower address), even if a VFP FPU is selected.
<code>/fpe2</code>		Similar to <code>/fpa</code> , this is the default setting if <code>--fpu=fpe2</code> is specified.
<code>/fpe3</code>		Similar to <code>/fpa</code> , this is the default setting for other FPA FPUs. This differs from <code>/fpe2</code> in that it sets the FP3 attribute on any code AREAs.
<code>/vfp</code>		This is the default setting if a VFP FPU is selected. Sets the VFP attribute for any code or data AREAs, and sets VFP endianness rules for double-precision floating point numbers (endianness matches the rest of the system), even if an FPA FPU is selected.
<code>/softfp</code>		Sets the SoftFP attribute by default on imported and exported code symbols. This implies a floating point calling standard which is compatible with software floating point libraries (floating point arguments and results are passed in integer registers and/or the stack).
<code>/hardfp</code>		Does not set the SoftFP attribute by default on imported and exported code symbols. This is the default setting.
<code>/fpregargs</code> or <code>/fpr</code>		Further qualifies <code>/hardfp</code> . Sets the <code>fp-arguments-in-fp-registers</code> attribute by default on imported and exported code symbols. This is the default setting when <code>/vfp</code> is specified.

Command line option	Short form	Description
<code>/nofpregargs</code> or <code>/nofpr</code>		Further qualifies <code>/hardfp</code> . Does not set the <code>fp-arguments-in-fp-registers</code> attribute by default on imported and exported code symbols, implying the use of a floating point calling standard where floating point results are passed in hardware FP registers, but floating point arguments are passed in integer registers and/or the stack. This is the default setting when <code>/fpa</code> is specified, because of the inefficiencies of using emulated instructions for argument marshalling, since the majority of CPUs use the FPEmulator.
<code>--depend=dependfile</code>	<code>-d</code>	Saves source file dependency lists, which are suitable for use with 'make' utilities.
<code>-m</code>		Like <code>--depend</code> , but prints dependency information to the screen instead of sending it to a file.
<code>--absolute</code>	<code>-abs</code>	Accepts AAsm source code to provide some backwards compatibility in this release. See the appendix <i>Support for AAsm source</i> on page 211.
<code>--predefine="directive"</code>	<code>-pd</code>	Allows you to set an initial value for an assembler global variable. You must give a valid variable name, followed by a <code>SETL</code> , <code>SETA</code> or <code>SETS</code> directive, followed by a value. The value may be a simple constant or a constant expression (in ObjAsm syntax) of appropriate type – logical, arithmetic or string for <code>SETL</code> , <code>SETA</code> and <code>SETS</code> respectively – provided that its value can be computed at the start of assembly. The variable is set as if the directive occurs before the start of the source; an implicit <code>GBLL</code> , <code>GBLA</code> or <code>GBLS</code> directive is also executed. In the case of <code>SETS</code> , quotation marks are usually necessary around the value, since it is a string expression.; these must be escaped by preceding each with a backslash ('\ <code>\</code> ').
<code>-from filename</code>	<code>-f</code>	Supported, for backward compatibility with previous release.

Command line option	Short form	Description
-to <i>filename</i>	-t	Supported, for backward compatibility with previous release.
-print «<i>listingfile</i>»	-p	Supported, for backward compatibility with previous release.
-closeexec	-c	Recognised but ignored, for backward compatibility with previous release.
-module		Recognised but ignored, for backward compatibility with previous release.
-quit	-q	Recognised but ignored, for backward compatibility with previous release.
-stamp	-s	Recognised but ignored, for backward compatibility with previous release.

Part 2 – Assembly language details

3

ARM assembly language

ARM Assembly Language is the language which ObjAsm parses and compiles to produce object code in ARM Object Format. Information on ObjAsm command line options are detailed in *ObjAsm command lines* on page 30. This chapter details ARM Assembly Language, but does not give examples of its use.

General

Instruction mnemonics and register names may be written in upper or lower case (but not mixed case). Directives must be written in upper case.

Input lines

The general form of assembler input lines is:

«label» «instruction» «;comment»

A space or tab should separate the label, where one is used, and the instruction. If no label is used the line must begin with a space or tab. Any combination of these three items will produce a valid line; empty lines are also accepted by the assembler and can be used to improve the clarity of source code.

Assembler source lines are allowed to be up to 4095 characters long. To make source files easier to read, a long line of source can be split onto several lines by placing a backslash character, '\', at the end of a line. The backslash must not be followed by any other characters (including spaces or tabs). The backslash + end of line sequence is treated by ObjAsm as white space. Note that the backslash + end of line sequence should not be used within quoted strings.

AREAs

AREAs are the independent, named, indivisible chunks of code and data manipulated by the Linker. The Linker places each AREA in a program image according to the AREA placement rules (i.e. not necessarily adjacent to the AREAs with which it was assembled or compiled).

Conventionally, an assembly, or the output of a compilation, consists of two AREAs, one for the code (usually marked read-only), and one for the data which may be written to. A reentrant object will generally mark its data AREA as

BASED sb (see below), which means it defines relocatable address constants. This allows the code area to be read-only, position-independent and reentrant, making it easily ROM-able. Sometimes a third AREA will be used for data which is initialised to zero. This is useful for reducing the binary size, since data initialisers for such AREAs do not need to be included.

In ARM assembly language, each AREA begins with an **AREA** directive. If the AREA directive is missing the assembler will generate an AREA with an unlikely name (|\$\$\$\$\$\$\$|) and produce a diagnostic message to this effect. This will limit the number of spurious errors caused by the missing directive, but will not lead to a successful assembly.

The syntax of the AREA directive is:

```
AREA name«,attr»«,attr»...
```

You may choose any name for your AREAs, but certain choices are conventional. For example, |C\$\$code| is used for code AREAs produced by the C compiler, or for code AREAs otherwise associated with the C library.

Area attributes

AREA attributes are as follows:

ABS	Absolute: rooted at a fixed address.
REL	Relocatable: may be relocated by the Linker (the default).
PIC	Position Independent Code: will execute where loaded without modification.
CODE	Contains machine instructions.
DATA	Contains data, not instructions.
READONLY	This area will not be written to.
READWRITE	The opposite of READONLY. This is the default setting.
COMDEF	Common area definition.
COMMON	Common area.
NOINIT	Data AREA initialised to zero: contains only space reservation directives, with no initialised values.
REENTRANT	The code AREA is reentrant. This can also be specified using the /reentrant switch on the --apcs command line option.

BASED <i>Rn</i>	where <i>Rn</i> is a register, conventionally R9 or sb . Defines a data area which is consolidated by the linker with other data areas sharing the same base register. Any label defined within this AREA becomes a register-relative expression which can be used with LDR , STR, ADR and related instructions. For full details see the appendix <i>ARM procedure call standard</i> on page 263 of the <i>Desktop Tools</i> guide.
ALIGN=<i>expression</i>	The ALIGN sub-directive forces the start of the area to be aligned on a power-of-two byte-address boundary. By default AREAs are aligned on a 4-byte word boundary, but the expression can have any value between 2 and 12 inclusive.
INTERWORK	Specifies that the source is built for ARM/Thumb interworking. Interworking can also be specified using the /interwork switch on the --apcs command line option.
HALFWORD	Indicates that the area is using halfword memory access. This should not normally be needed as the assembler will detect their use automatically.
NOSWSTACKCHECK	Indicates that a code area does not use stack limit checking. This can also be specified using the /noswstackcheck switch on the --apcs command line option.
VFP	Double-precision floating point data, arguments and results are stored using VFP rather than FPA endianness rules. This can also be specified using the /vfp switch on the --apcs command line option.
CODEALIGN	Allows certain types of ALIGN directives in a code area to pad using NOP instructions. See <i>Miscellaneous directives – ALIGN, NOPF, RLIST and ENTRY</i> on page 72

ORG and ABS

ORG *base-address*

The ORG (origin) directive is used to set the base address and the ABS (absolute) attribute of the containing AREA, or of the following AREA if there is no containing AREA. In some circumstances this will create objects which cannot be linked. In general it only makes sense to use ORG in programs consisting of one AREA, which need to map fixed hardware addresses such as trap vector locations. Otherwise ORG should be avoided.

Instruction sets and syntax

ObjAsm supports ARM and Thumb instruction sets, and UAL and traditional (pre-UAL) assembler syntax. You can control this from the command line (see page 31) or insert one of the following directives in the source code:

Directive	Instruction set	Syntax
CODE32	ARM	Pre-UAL
CODE16	Thumb	Pre-UAL
ARM	ARM	UAL

Multiple such directives may be present in the same source file, allowing a mixture of ARM and Thumb code. Note that the linker does **not** currently support generation of ARM/Thumb interworking veneers, so calls to external areas must use B, BL, BLX or BX depending on the type of the external symbol.

When you write ARM code, ObjAsm will accept either pre-UAL or UAL instructions irrespective of which directive is in force, but will warn you if the instruction is incorrect in the selected syntax. However, when you write Thumb code, accepting both syntaxes like that is impossible because in many cases, the same textual representation maps to differing instructions, and so ObjAsm requires strict conformance to the selected syntax.

Object file format

ObjAsm supports two object file formats, AOF (the default) and a.out. This can be set explicitly by using one of the following directives:

AOF

AOUT

on a line by itself. Only one of these directives may be present in each source file.

Symbols

Numbers, logical values, string values and addresses may be represented by symbols. Symbols representing numbers or addresses, logical values and strings are declared using the GBL and LCL directives, and values are assigned immediately by SETA, SETL and SETS directives respectively (see *Local and global variables* – GBL, LCL and SET on page 74). Addresses are assigned by the Assembler as assembly proceeds, some remaining in symbolic, relocatable form until link time.

Symbols must start with a letter in either upper or lower case; the assembler is case-sensitive and treats the two forms as distinct. Numeric characters and the underscore character may be part of the symbol name. All characters are significant.

Symbols should not use the same name as instruction mnemonics or directives. While the assembler can distinguish between the uses of the term through their relative positions in the input line, a programmer may not always be able to do so.

Symbol length is limited by the 4095 character line length limit.

If there is a need to use a wider range of characters in symbols, for instance when working with other compilers, use enclosing bars to delimit the symbol name; for example, `|C$$code|`. The bars are not part of the symbol.

Labels

Labels are a special form of symbol, distinguished by their position at the start of lines. The address represented by a label is not explicitly stated but is calculated during assembly.

If a label appears on a line with an instruction or a data-allocating directive (like DCD), then the label will be given a CODE or DATA attribute respectively, unless explicitly overridden in an EXPORT directive.

In older releases of ObjAsm, labels on a line by themselves were taken to be CODE labels in code AREAs, or DATA labels in data AREAs; the only way to override this was by use of the DATA directive:

```
label DATA
```

This allocates a DATA label without emitting any bytes, whether it is in a code AREA or a data AREA.

In ObjAsm 4, labels on lines by themselves default to DATA, even in code AREAs, if they follow a directive that outputs data. For example, the label below has changed from CODE to DATA:

```

          DCD      -1
label
          MOV      pc, r14
```

To ensure the label is exported as code in both old and new versions of ObjAsm, either define the label on the same line as an opcode, or precede it with a ROUT directive.

Local labels

The local label, a subclass of label, begins with a number in the range 0-99. Local labels work in conjunction with the ROUT directive and are most useful for solving the problem of macro-generated labels. Unlike global labels, a local label may be defined many times; the assembler uses the definition closest to the point of reference. To begin a local label area use:

<label> ROUT

The label area will start with the next line of source, and will end with the next ROUT directive or the end of the program.

Local labels are defined as:

number<routineName>

although **routineName** need not be used; if omitted, it is assumed to match the label of the last ROUT directive. It is an error to give a routine name when no label has been attached to the preceding ROUT directive.

References to local labels

A reference to a local label has the following syntax:

%<x><y>n<routineName>

% introduces the reference and may be used anywhere where an ordinary label reference is valid.

x tells the assembler where to search for the label; use **B** for backward or **F** for forward. If no direction is specified the assembler looks both forward and backward. However searches will never go outside the local label area (i.e. beyond the nearest ROUT directives).

y provides the following options: **A** to look at all macro levels, **T** to look only at this macro level, or, if **y** is absent, to look at all macro from the current level to the top level.

n is the number of the local label.

routineName is optional, but if present it will be checked against the enclosing ROUT's label.

Comments

The first semi-colon on a line marks the beginning of a comment, except where the semi-colon appears inside a string constant. A comment alone is a valid line. An asterisk is also a valid comment character if it appears as the first character of a line. All comments are ignored by the assembler.

Constants

Numbers

Numeric constants are accepted in three forms: decimal (e.g. **123**), hexadecimal (e.g. **&7B** or **0x7B**), and **n_xxx**, where **n** is a base between 2 and 9, and **xxx** is a number in that base.

Strings

Strings consist of opening and closing double quotes, enclosing characters and spaces. If double quotes or dollar signs are used within a string as literal text characters, they should be represented by a pair of the appropriate character; e.g. **\$\$** for **\$**.

Boolean

The Boolean constants 'true' and 'false' should be written as **{TRUE}** and **{FALSE}**.

The END directive

Every assembly language source must end with:

END

on a line by itself.

4 CPU instruction set

For up-to-date information on the ARM and Thumb instruction sets, we recommend the ARM Architecture Reference Manual, see *Recommended Books* on page 3.

ObjAsm understands a number of other instructions, which it translates into appropriate basic ARM and Thumb instructions.

Extended range immediate constants

Synopsis

In the case of an instruction such as

```
MOV    R0,#constant
```

ObjAsm will evaluate the expression and produce a CPU instruction to load the value into the destination register. This may not in fact be the machine level instruction known as MOV, but the programmer need not be aware that an alternative instruction has been substituted. A common example is

```
MOV    Rn,#-1
```

which the CPU cannot handle directly (as -1 is not a valid immediate constant). ObjAsm will accept this syntax, but will convert it and generate object code for

```
MVN    Rn,#0
```

which results in R_n containing -1 . Such conversions also takes place between the following pairs of instructions:

- **BIC/AND**
- **ADD/SUB**
- **ADC/SBC**
- **CMP/CMN**

When assembling with `--cpu=6T2` or later, ObjAsm can also convert MOV and MVN immediate instructions into MOVW instructions, to further extend the range of supported immediate constants.

The MOV32 instruction

Assembler syntax

MOV32*<cond>* *register*,#*constant*

MOV32*<cond>* *register*,*expression*

Synopsis

This pseudo-instruction assembles to a pair of instructions, MOVW and MOVH, both of which are available in ARMv6T2 or later. It enables an arbitrary 32-bit number to be loaded into a register, without danger of polluting the data cache by using a literal pool. Even relocatable expressions can be used as a parameter - the linker knows how to rewrite the instructions when linking, and there is also support in the relocation code it outputs for modules and relocatable AIF executables to support load-time relocation of this pseudo-instruction.

The ADR instruction

Assembler syntax

ADR*<cond>* *register*,*expression*

Synopsis

This produces an address in a register. ARM does not have an explicit 'calculate effective address' instruction, as this can generally be done using ADD, SUB, MOV or MVN. To ease the construction of such instructions, ObjAsm provides an ADR instruction.

The expression may be register-relative, program-relative or numeric:

- **Register-relative:** **ADD|SUB** *register*,*register2*,#*constant*
will be produced, where *register2* is the register to which the expression is relative.
- **Program-relative:** **ADD|SUB** *register*,PC,#*constant*
will be produced.
- **Numeric:** **MOV|MVN** *register*,#*constant*
will be produced. When assembling with **--cpu=6T2** or later, ObjAsm may output a MOVW instruction if the instruction cannot be accomplished using a MOV or MVN.

In all three cases, an error will be generated if the immediate constant required is out of range.

If the program has a fixed origin (that is, if the ORG directive has been used), the distinction between program-relative and numeric values disappears. In this case, ObjAsm will first try to treat such a value as program-relative. If this fails, it will try to treat it as numeric. An error will only be generated if both attempts fail.

The ADRL instruction

Assembler syntax

ADR«*cond*»**L** *register, expression* (*pre-UAL syntax*)

ADRL«*cond*» *register, expression* (*UAL syntax*)

Synopsis

This form of ADR is provided by ADRL and allows a wider collection of effective addresses to be produced. ADRL can be used in the same way as ADR, except that the allowed range of constants is larger. Again program-relative, register relative and numeric forms exist. When assembling with **--cpu=6T2** or later, ObjAsm will output a MOV32 pseudo-instruction for the numeric form if the instruction cannot be accomplished using MOV, ADD or MVN, SUB. The result produced will always be two instructions, even if it could have been done in one. An error will be generated if the necessary immediate constants cannot be produced.

The IT instruction

Assembler syntax

IT«**T**|**E**«**T**|**E**«**T**|**E**»»» *cond*

Synopsis

IT is a valid Thumb instruction in ARMv6T2 or later - see the ARM ARM for a full description. But it can also be used in ARM code as a pseudo-instruction, in which case it doesn't emit any code itself, but it does perform condition code checking on the following 1-4 instructions, just as it would have done had it been used in Thumb code.

The UND instruction

Assembler syntax

UND*<cond>* *<#constant>*

Synopsis

This pseudo-instruction gives a way to access the instructions which have been reserved to remain as undefined instructions in all future ARM architectures. For ARM instructions, the constant must be in the range 0-65535 - if omitted, it defaults to 0.

Literals

Assembler syntax

LDR*<cond>***B** *register*,=*expression* (*pre-UAL syntax*)

LDRB*<cond>* *register*,=*expression* (*UAL syntax*)

LDR*<cond>***SB** *register*,=*expression* (*pre-UAL syntax*)

LDRSB*<cond>* *register*,=*expression* (*UAL syntax*)

LDR*<cond>***H** *register*,=*expression* (*pre-UAL syntax*)

LDRH*<cond>* *register*,=*expression* (*UAL syntax*)

LDR*<cond>***SH** *register*,=*expression* (*pre-UAL syntax*)

LDRSH*<cond>* *register*,=*expression* (*UAL syntax*)

LDR *register*,=*expression*

LDR*<cond>***D** *register*,=*expression* (*pre-UAL syntax*)

LDRD*<cond>* *register*,=*expression* (*UAL syntax*)

Synopsis

Literals are intended to enable the programmer to load immediate values into a register which might be out of range as MOV/MVN arguments.

ObjAsm will take certain actions with literals. It will:

- if possible, replace the instruction with a MOV or MVN. When assembling with `--cpu=6T2` or later, ObjAsm may output a MOVW instruction if the instruction cannot be accomplished using a MOV or MVN.

- otherwise, generate a program-relative load instruction, and if no such literal already exists within the addressable range, place the literal in the next literal pool. In the case of LDRD, the literal is allocated doubleword-aligned within the literal pool, because some CPUs (such as the XScale) require this alignment.

Program-relative expressions and imported symbols are also valid literals. See the section *Organisational directives – END, ORG, LORG, KEEP and LEAF* on page 66 for further information.

Shifts by zero

All ALU instructions which specify an immediate shift of 0 are converted into **LSL #0** (i.e. unshifted) forms.

5 Floating point instructions

ARM cores supports up to 16 coprocessors, and most implementations include one or more of them. Coprocessors are most commonly used to implement floating point and/or SIMD instructions. A subsection of the ARM instruction set is reserved for the use of each coprocessor, and from architecture 6T2 onwards, these coprocessor instructions are also available in the Thumb instruction set.

The most notable coprocessors are the following:

Coprocessor	Data types supported	Coprocessor number(s)
FPA	single, double and extended precision floating point	1, 2
System control	-	15
Debug	-	14
MaverickCrunch	single and double precision floating point; 32 and 64-bit integer	4, 5, 6
XScale CP0	40-bit integer	0
Wireless MMX	64-bit integer SIMD	0, 1
VFP	single and double precision floating point; 64 and 128-bit integer SIMD (status and data transfer instructions only)	10, 11

Any instructions for a coprocessor which is not fitted take the undefined instruction trap. This makes it possible for a software emulator to be written which will simulate the coprocessor and allow software written for a coprocessor to execute on a processor which lacks that coprocessor, albeit at a greatly reduced speed. To date, this has only been done for the FPA, via the floating point emulator, which has been incorporated into RISC OS as the FP emulator module. Generally, programs do not need to know whether a coprocessor is fitted; the only effective difference is in the speed of execution. Note that there may be slight variations in accuracy between hardware and software – refer to the instructions supplied with the coprocessor for details of these variations.

Despite the speed penalty, the default method of handling floating point on RISC OS has always been to use FPA instructions and rely on this mechanism to execute them on the majority of ARM CPUs where there is no FPA hardware. As a result of the reliance on software emulation, FPA instructions could traditionally only be executed from user mode. This restriction was lifted in RISC OS 4.

Coprocessor instructions can always be expressed using the generic coprocessor instructions CDP, CDP2, LDC, LDC2, STC, STC2, MCR, MCR2, MCRR, MCRR2, MRC, MRC2, MRRC and MRRC2. But these are usually not very human-readable, so alternative syntaxes are usually devised which more closely reflect the instruction's functionality. ObjAsm has support for alternative syntaxes for the following coprocessors:

- FPA (see the 7500FE data sheet)
- XScale coprocessor 0 (see the XScale core manual)
- VFP (see the ARM ARM), both pre-UAL and UAL syntaxes

These documents are described in *Recommended Books* on page 3.

This chapter also covers a few features of ObjAsm's support for the Advanced SIMD extension (also known as NEON, which is technically the name of ARM's own implementation of the extension). Although it is not, in the main part, implemented using the coprocessor interface, its close relationship to the VFP means that it is sensible to discuss it here.

Floating point constants

Wherever ObjAsm accepts a floating point number, any one of a variety of formats are accepted:

- **`<<+ | ->mantissa<<E | e<<+ | ->exponent>`**
The mantissa part consists of a sequence of zero or more decimal digits, followed by an optional decimal point followed by a sequence of zero or more decimal digits. The mantissa must contain a non-zero number of digits overall. The exponent part consists of a sequence of one or more decimal digits. The value generated represents the mantissa multiplied by ten to the power of the exponent, where the exponent is taken to be zero if missing. All reading is done to double precision, with overflows resulting in infinities, and is then narrowed if required.
- **`<<+ | ->base_mantissa<<E | e<<+ | ->exponent>`**
The base part is a single decimal digit in the range 2 to 9. The mantissa part is similar to the mantissa for the decimal case, but may only use valid digits for the specified base. The exponent consists of one or more decimal digits. The value generated represents the mantissa multiplied by the base to the power

of the exponent, where the exponent is taken to be zero if missing. All reading is done to double precision, with overflows resulting in infinities, and is then narrowed if required.

- `<<+ | ->&mantissa<<P | p<<+ | ->exponent>>`
`<<+ | ->0xmantissa<<P | p<<+ | ->exponent>>`

The mantissa part is similar to the mantissa for the decimal case, but may use any (case-insensitive) hexadecimal digit. The exponent consists of one or more decimal digits. The value generated represents the mantissa multiplied by two (not sixteen) to the power of the exponent, where the exponent is taken to be zero if missing. All reading is done to double precision, with overflows resulting in infinities, and is then narrowed if required.

- **`0f_hexdigits`**

This method lets you specify a floating point number as an eight-digit hexadecimal number corresponding to the number's representation as a single-precision floating point number. The number is widened or narrowed if required.

- **`0d_hexdigits`**

This method lets you specify a floating point number as a sixteen-digit hexadecimal number corresponding to the number's representation as a single-precision floating point number. The number is narrowed if required.

Examples are:

```
1
0.2
5E9
E-2
-.7
+31.415926539E-1
2_1100.1001e+6
+8_.7
-0Xfff.fffffP-12
0F_7f7fffff
0d_FEDCBA9876543210
```

Extended range immediate constants

Synopsis

ObjAsm has analogous support for VFP and Advanced SIMD immediate constants to that for ARM immediate constants (described in *Extended range immediate constants* on page 49). Specifically, if

VMOV«*cond*». *type register, #constant*

cannot be expressed using any of the available immediate constant encodings, ObjAsm will instead try to encode it using a VMVN instruction and the binary NOT of the constant, and vice versa.

type can be any of I8, I16, I32, I64, F16, F32 or F64, except when the destination register is a single-precision floating point register, when only I32 or F32 are available. *constant* will be interpreted and range-checked as a floating point or integer number (and as a signed or unsigned integer if necessary) to match the specified data type. In some cases where constant cannot be achieved using the specified type, ObjAsm can substitute an alternative instruction which loads the required bit pattern into the register, but expresses it using a differing combination of *type* and *constant* that is permitted by the instruction set.

In a similar vein, *type* and *constant* can be substituted in VBIC and VORR to allow otherwise impossible constants to be used. And although the Advanced SIMD instruction set does not natively support variants of VAND and VORN which take an immediate constant, ObjAsm will construct these from VBIC or VORR respectively, again by NOTting the constant.

ObjAsm will, if necessary and possible, round single and double-precision floating point constants to the nearest valid immediate constant. A warning is emitted if this happens.

The VMOV2 instruction

Assembler syntax

VMOV2«*cond*». *type Dd, #constant*

VMOV2«*cond*». *type Qd, #constant*

Synopsis

This pseudo-instruction assembles to a pair of instructions, a combination of VMOV or VMVN and VORR and VBIC. This enables a wider range of constants to be loaded into all elements of the specified vector than would be possible using a single VMOV or VMVN instruction. The same values for *type* are permitted as for the VMOV (constant) instruction.

Register comparison instructions

Assembler syntax

```
V<A>CLE<cond>.type <Dd,>>Dn,Dm
```

```
V<A>CLE<cond>.type <Qd,>>Qn,Qm
```

```
V<A>CLT<cond>.type <Dd,>>Dn,Dm
```

```
V<A>CLT<cond>.type <Qd,>>Qn,Qm
```

Synopsis

The Advanced SIMD instruction set does not include these instructions natively. Instead, ObjAsm achieves these by exchanging the source registers and encoding a greater-than instruction in place of a less-than instruction, or a greater-than-or-equal instruction in place of a less-than-or-equal instruction.

2 x 32-bit vector zip and unzip

Assembler syntax

```
VUZP<cond>.32 Dd,Dm
```

```
VZIP<cond>.32 Dd,Dm
```

Synopsis

These two instructions are equivalent in functionality, but the Advanced SIMD instruction set does not include either of them natively. Instead, ObjAsm assembles them both to

```
VTRN<cond>.32 Dd,Dm
```

which has the same effect as you would have expected from the VUZP or VZIP instructions.

Literals

Assembler syntax

```
LDF<<cond>>S Fn, =floating point constant
LDF<<cond>>D Fn, =floating point constant
FLDS<<cond>> Sn, =floating point constant
FLDD<<cond>> Dn, =floating point constant
VLDR<<cond>><<.type>> Sn, =floating point or integer constant
VLDR<<cond>><<.type>> Dn, =floating point or integer constant
```

Synopsis

For VLDR, *type* can be I8, I16, I32, I64, F16, F32 or F64, except that the 64-bit types are not available if the destination register is 32 bits wide. Data smaller than the register are replicated to fill the register.

Coprocessor literal handling is similar to that for the main ARM instruction set (see *Literals* on page 52). ObjAsm will

- if possible, replace an LDF with an MVF or MNF
- if possible, replace an FLD/VLDR with an FCONST/VMOV
- otherwise, place the constant in the nearest literal pool if necessary, and construct a PC-relative LDC instruction to load the destination register from the literal pool. The endianness of any double-precision floating point value in the literal pool is set according to the instruction which references it, irrespective of the current **--apcs /vfp** flag. Because the allowed offset range within an LDC instruction is less than that for a LDR instruction (–1020 to +1020 instead of –4095 to +4095), it may be necessary to code LTORG directives more frequently if floating point literals are being used than would otherwise be necessary.

Right shifts by zero

The Advanced SIMD instruction set includes a number of right-shift instructions, but unlike the left-shift instructions, these cannot encode an immediate shift by 0. If you specify such a shift, ObjAsm will instead substitute an alternative instruction which has the same narrowing and saturation features as the shift instruction it replaces:

Instruction	Replacement
V<R>SRA <i>d,m,#0</i>	VADD <i>d,m</i>
VSRI <i>d,m,#0</i>	VMOV <i>d,m</i>
V<R>SHR <i>d,m,#0</i>	VMOV <i>d,m</i>
V<R>SHRN <i>d,m,#0</i>	VMOVN <i>d,m</i>
VQ<R>SHRN <i>d,m,#0</i>	VQMOVN <i>d,m</i>
VQ<R>SHRUN <i>d,m,#0</i>	VQMOVUN <i>d,m</i>

Fixed point conversions with zero fractional bits

The VFP and Advanced SIMD instruction sets include instructions for conversion, amongst other things, between single precision floating point and 32-bit fixed point numbers (1, 2 or 4 such conversions in parallel). The number of fractional bits in the fixed point number can normally be specified between 1 and 32. However, you can specify 0 fractional bits, and ObjAsm will substitute the equivalent instruction to convert to or from a 32-bit integer:

Instruction	Replacement
VCVT.F32.S32 <i>d,m,#0</i>	VCVT.F32.S32 <i>d,m</i>
VCVT.F32.U32 <i>d,m,#0</i>	VCVT.F32.U32 <i>d,m</i>
VCVT.S32.F32 <i>d,m,#0</i>	VCVT.S32.F32 <i>d,m</i>
VCVT.U32.F32 <i>d,m,#0</i>	VCVT.U32.F32 <i>d,m</i>

Note that conversion instructions to and from 16-bit fixed point numbers already support 0 fractional bits natively. Note also that conversions between double-precision floating point and 32-bit fixed point numbers cannot be substituted in this way

Unsigned saturation of signed numbers

The Advanced SIMD instruction set includes a number of instructions which can perform an unsigned saturation on a signed number. These are indicated by a 'U' flag in the opcode in standard UAL syntax. However, ObjAsm also allows you to

specify this by using a pair of differing data type qualifiers (or, more usefully, by using register name symbols which have been declared with those types using the DN or QN directives).

Instruction	Replacement
<code>VQSHL.Usize.Ssize d,m,#imm</code>	<code>VQSHLU.Ssize d,m,#imm</code>
<code>VQMOVN.Usize.Ssize d,m</code>	<code>VQMOVUN.Ssize d,m</code>
<code>VQ<R>SHRN.Usize.Ssize d,m,#imm</code>	<code>VQ<R>SHRUN.Ssize d,m,#imm</code>

6 Directives

This chapter describes the directives available in the assembler, which provide a powerful range of extra features.

Storage reservation and initialisation – DCB, DCW, DCD etc

DCB	Defines one or more bytes: can be replaced by =
DCW	Defines one or more half-words (16-bit numbers)
DCWU	Defines one or more half-words at arbitrary alignment
DCD	Defines one or more words: can be replaced by &
DCDU	Defines one or more words at arbitrary alignment
DCQ	Defines one or more 64-bit words
DCQU	Defines one or more 64-bit words at arbitrary alignment
DCDO	Defines one or more words consisting of offsets from a base register
DCI	Defines one or more words or half-words, marking them as code
SPACE	Reserves a zeroed area of store: can be replaced by %
FILL	Reserves an area of store with a specified initialiser

The syntax of the first eight directives is:

«label» directive expression-list

DCD can take program-relative and external expressions as well as numeric ones.

An *external expression* includes one or more references to symbols from another source file or from another area of the same source file. These are achieved by the insertion of relocations into the object file. Prior to ObjAsm 4, the allowed syntax of external expressions was far more restrictive: you could only use a single symbol, optionally offset by a constant expression.; the external symbol had to come first.

In the case of DCB, the *expression-list* can include string expressions, the characters of which are loaded into consecutive bytes in store. Unlike C strings, ObjAsm strings do not contain an implicit trailing NUL, so a C string has to be fabricated thus:

```
C_string DCB "C_string",0
```

The DCDO directive defines one or more words, like DCD, but is intended for storing the offsets to labels in BASED AREAs from their base register. Different relocations are output in order to instruct the linker to adjust the number to account for the merging of all the AREAs which share the same base register. As of ObjAsm 4, the effect of

```
DCD :INDEX: symbol
```

is equivalent to that of

```
DCDO symbol
```

and this may help you understand what it does.

The syntax of DCI is:

```
<label> DCI<.N|.W> expression-list
```

This is similar to DCW and DCD but marks the words in the object file as code rather than data. This allows them to be correctly interpreted by disassemblers and debuggers. The optional .N or .W suffix - this behaves like the instruction width specifier in UAL syntax, and determines whether the instruction allocated is 16 (the default) or 32 bits wide in Thumb code. Only the .W suffix is permitted in ARM code.

The syntax of SPACE is:

```
<label> SPACE size
```

This directive will initialise to zero the number of bytes specified by the numeric expression *size*.

The syntax of FILL is:

```
<label> FILL size,value,width
```

This acts like the SPACE directive, but fills the allocated space with the specified value. If *value* is omitted, it defaults to 0. *width* specifies the size of *value* in bytes, and can take the values 1, 2, or 4; if omitted, it defaults to 1.

Binary file inclusion – BIN and INCBIN

The following two directives:

BIN *filename*

INCBIN *filename*

are equivalent, and serve to insert the contents of the specified file at the current point within the object file. INCBIN is preferred, because it is also supported by armasm.

Floating point store initialisation – DCFH, DCFS and DCFD

DCFH	Defines half precision floating point values
DCFHU	Defines half precision floating point values at arbitrary alignment
DCFS	Defines single precision floating point values
DCFSU	Defines single precision floating point values at arbitrary alignment
DCFD	Defines double precision floating point values
DCFDU	Defines double precision floating point values at arbitrary alignment

The syntax of these directives is:

«label» directive *fp-constant*«, *fp-constant*»

Half precision numbers occupy one half-word (16 bits) and are half-word aligned by default. DCFH and DCFHU accept numbers which can be expressed either as IEEE or VFP's alternate half precision format and selects the appropriate format automatically.

Single precision numbers occupy one word, and double precision numbers occupy two; both are word aligned by default. The endianness of numbers stored by DCFD and DCFDU depends on the **--apcs /vfp** option.

See *Floating point constants* on page 56 for the acceptable formats for *fp-constant*.

Describing the layout of store – MAP and FIELD

MAP Sets the origin of a storage map: can be replaced by **^**
FIELD Reserves space within a storage map: can be replaced by **#**

The syntax of these directives is:

```
          MAP expression«,base-register»  
«symbol» FIELD expression
```

The MAP directive sets the origin of a storage map at the address specified by the expression. A storage map location counter, @, is also set to the same address. The expression must be fully evaluable in the first pass of the assembly, but may be program-relative or relative to a symbol in another area in the same file. If no MAP directive is used, the @ counter is set to zero. @ can be reset any number of times using MAP to allow many storage maps to be established.

Space within a storage map is described by the FIELD directive. Every time FIELD is used its label (if any) is given the value of the storage location counter @, and @ is then incremented by the number of bytes reserved.

In a MAP directive with a base register, the register becomes implicit in all symbols defined by FIELD directives which follow, until cancelled by a subsequent MAP directive. These register-relative symbols can later be quoted in load and store instructions. For example:

```
          ^ 0,r9  
          # 4  
Lab      # 4  
          LDR r0,Lab
```

is equivalent to:

```
LDR r0,[r9,#4]
```

Organisational directives – END, ORG, LTORG, KEEP and LEAF

END

The assembler stops processing a source file when it reaches the END directive. If assembly of the file was invoked by a GET directive, the assembler returns and continues after the GET directive (see *Links to other source files – GET/INCLUDE* on page 69). If END is reached in the top-level source file during the first pass without any errors, the second pass will begin. Failing to end a file with END is an error.

ORG *numeric-expression*

A program's origin is determined by the ORG directive, which sets the initial value of the program location counter. Only one ORG is allowed in an assembly and no ARM instructions or store initialisation directives may precede it. If there is no ORG, the program is relocatable and the program counter is initialised to 0.

LTORG

LTORG directs that the current literal pool be assembled immediately following it. A default LTOrg is executed at every END directive which is not part of a nested assembly, but large programs may need several literal pools, each closer to where their literals are used to avoid violating LDR's 4KB offset limit.

KEEP «*symbol*»

The assembler does not by default describe local symbols (i.e. *non-exported* symbols; see *Links to other object files – IMPORT, EXPORT, etc* on page 67) in its output object file. However, they can be retained in the object file's symbol table by using the KEEP directive. If the directive is used alone all symbols are kept; if only a specific symbol needs to be kept it can be specified by name.

LEAF *symbol*

ensures the AOF LEAF attribute is set on symbol. Like KEEP, it also forces symbol to appear in the symbol table (as a local symbol) if it is not otherwise mentioned in an EXPORT or KEEP directive.

The KEEP and LEAF directives cannot override the floating point calling standard symbol attributes set by the --apcs qualifiers.

Links to other object files – IMPORT, EXPORT, etc

```
IMPORT symbol «[qualifier-list]»«,WEAK»
IMPORT symbol«,FPREGARGS»«,WEAK» (deprecated)
```

```
EXPORT symbol «[qualifier-list]»
EXPORT symbol«,symbol ...» (deprecated)
```

EXPORT

IMPORT provides the assembler with a name (symbol) which is not defined in this assembly, but will be resolved at link time to a symbol defined in another, separate object file. The **EXTERN** directive is similar in syntax and operation to IMPORT, but it differs in that the symbol is only inserted into the object file as an external reference if the symbol is actually referenced in the source file - this means it works like the 'C' **extern** keyword.

EXPORT declares a symbol for use at link time by other, separate object files. The **GLOBAL** directive is an alias for EXPORT. If EXPORT is used with no parameters, it exports all local labels to the object file, similar to the parameter-less form of the KEEP directive.

qualifier-list is a comma-separated list of qualifiers from the following list:

Qualifier	Allowed in	Effect
FPREGARGS	Both	Defines a function which expects floating point arguments passed to it in floating point registers.
NOFPREGARGS	Both	Opposite of FPREGARGS. The default setting is determined by the --apcs switch, so this may be needed as an override.
SOFTFP	Both	Defines a function which follows a SoftFP floating point calling standard.
HARDFP	Both	Opposite of SOFTFP. The default setting is determined by the --apcs switch, so this may be needed as an override.
DATA	Both	Defines a data location.
CODE	Both	Opposite of DATA. Defines a function.
ARM	EXPORT	Defines a function which should be entered in ARM mode. This is the default for any code symbols following a CODE32 or ARM directive.
THUMB	EXPORT	Defines a function which should be entered in Thumb mode. This is the default for any code symbols following a CODE16 directive.
LEAF	EXPORT	Defines a leaf function which calls no other functions.
NONLEAF	EXPORT	The opposite of LEAF.
USESSB	EXPORT	Sets symbol attribute bit 10, which is not currently used by the linker.
NOUSESSB	EXPORT	The opposite of USESSB.
WEAK	IMPORT	The linker will not fault an unresolved reference to this symbol, but will zero the location referring to it. This qualifier can also be placed outside the brackets for backwards compatibility.
NOWEAK	IMPORT	The opposite of WEAK.
READONLY	IMPORT	Indicates that the symbol is from a READONLY AREA.

Qualifier	Allowed in	Effect
READWRITE	IMPORT	Indicates that the symbol is from a READWRITE AREA.
BASED Rn	IMPORT	Indicates that the symbol is an offset from the specified base register.

If `--apcs=/rwpi` is specified, then any IMPORTed symbols with both DATA and READWRITE attributes will automatically be treated by ObjAsm as relative to sb, unless they also have a differing BASED attribute. This permits the same source file to be assembled as either /rwpi or /norwpi with no code changes.

EXPORTAS *symbol, alias*

EXPORTAS allows a symbol *symbol* to be given a different name, *alias*, in the object file (and therefore as seen by all other source files) from the one used within the current source file.

REQUIRE *symbol*

REQUIRE inserts a reference to *symbol* without emitting any bytes to the current AREA. An error is generated if the symbol is not satisfied by a local definition or an IMPORT or EXTERN directive. This directive is most useful in conjunction with the EXTERN directive.

STRONG *symbol*

STRONG both declares *symbol* as a strong symbol and defines the current location as the value to which references to *symbol* from other object files will resolve. You can also use IMPORT on the same symbol if you need to refer to the non-strong occurrence of *symbol* earlier in the same source file, or if you want to define additional attributes of *symbol* (for example the DATA attribute).

Links to other source files – GET/INCLUDE

GET *filename*

INCLUDE *filename*

GET includes a file within the file being assembled. This file may in turn use GET directives to include further files. Once assembly of the included file is complete, assembly continues in the including file at the line following the GET directive. INCLUDE is a synonym for GET.

Diagnostic generation – ASSERT, ! and INFO

```
ASSERT logical-expression  
! is-error, string-expression «,is-warning»  
INFO is-error, string-expression «,is-warning»
```

ASSERT supports diagnostic generation. If the *logical expression* returns {FALSE}, a diagnostic is generated during the second pass of the assembly. ASSERT can be used both inside and outside macros.

! and INFO are related to ASSERT but are inspected on both passes of the assembly, providing a more flexible means for creating custom error messages. The arithmetic expression *is-error* is evaluated; if it does not equal zero, the string is printed as an error diagnostic and the assembly halts after pass one.

If *is-error* equals zero, no action is taken during pass one, but the string is printed during pass two. If the optional arithmetic expression *is-warning* is present and evaluates to a non-zero value, then the string is printed as a warning. Otherwise, the string is printed as a plain informational message. INFO differs slightly from ! in this last case, in that it outputs the line at which the directive was found as well.

Dynamic listing options – OPT

The OPT directive is used to set listing options from within the source code, providing that listing is turned on. The default setting is to produce a normal listing including the declaration of variables, macro expansions, call-conditioned directives and MEND directives, but without producing a pass one listing. These settings can be altered by adding the appropriate values from the list below, and using them with the OPT directive as follows:

OPT <i>n</i>	Effect
1	Turns on normal listing.
2	Turns off normal listing.
4	Page throw: issues an immediate form feed and starts a new page.
8	Resets the line number counter to zero.
16	Turns on the listing of SET, GBL and LCL directives.
32	Turns off the listing of SET, GBL and LCL directives.
64	Turns on the listing of macro expansions.
128	Turns off the listing of macro expansions.
256	Turns on the listing of macro calls.
512	Turns off the listing of macro calls.
1024	Turns on the pass one listing.
2048	Turns off the pass one listing.
4096	Turns on the listing of conditional directives.
8192	Turns off the listing of conditional directives.
16384	Turns on the listing of MEND directives.
32768	Turns off the listing of MEND directives.

Titles – TTL and SUBT

Titles can be specified within the code using the TTL (title) and SUBT (subtitle) directives. Each is used on all pages until a new title or subtitle is called. If more than one appears on a page, only the latest will be used: the directives alone create blank lines at the top of the page. The syntax is:

TTL *title*
SUBT *subtitle*

Miscellaneous directives – ALIGN, NOFP, RLIST and ENTRY

ALIGN *<power-of-two<,offset<,value<,width>>>>*

After store-loading directives have been used, the program counter (PC) will not necessarily point to a word boundary. If an instruction mnemonic is then encountered, the assembler will insert up to three bytes of zeros to achieve alignment. However, an intervening label may not then address the following instruction. If this label is required, ALIGN should be used. On its own, ALIGN sets the instruction location to the next word boundary. The optional *power-of-two* parameter – which is given in bytes – can be used to align with a coarser byte boundary, and the *offset* parameter to define a byte offset from that boundary.

value and *width*, if present, operate similarly to the FILL directive (see *Storage reservation and initialisation* – DCB, DCW, DCD *etc* on page 63), except that *width* defaults to 2 in Thumb code and 4 in ARM code. If they are omitted, and the ALIGN follows an instruction in an AREA with the CODEALIGN attribute, and *offset* is an integer multiple of the instruction width, then the space is padded with NOP instructions.

NOFP

In some circumstances there will be no support in either target hardware or software for floating point instructions. In these cases the NOFP directive can be used to ensure that no floating point instructions or directives are allowed in the code.

RLIST

The syntax of this directive is:

label RLIST list-of-registers

The RLIST (register list) directive can be used to give a name to a set of registers to be transferred by LDM, STM, POP or PUSH. *List-of-registers* is a list of register names or ranges enclosed in **{ }**.

ENTRY

The ENTRY directive declares its offset in its containing AREA to be the unique entry point to any program containing this AREA.

7

Symbolic capabilities

The assembler also has a range of symbolic capabilities, with which you can set up symbols as constants or as variables. These are described below.

Setting constants

EQU is used to give a symbolic name to a constant or an address, which may be an absolute address, a program-relative address, or an address within another AREA from the same file. The ***** directive is an alias for EQU. The syntax is

symbol EQU expression«***,attribute***»

The *attribute* field can be present if *expression* is a constant expression whose value is known in pass 1. This allows for definition of the code/data and ARM/Thumb attributes of absolute addresses:

Attribute	Effect
ARM or CODE32	Defines an absolute ARM code label
THUMB or CODE16	Defines an absolute Thumb code label
DATA	Defines an absolute data label

RN defines register names. Registers can only be referred to by name. The names **R0-R15**, **r0-r15**, **SP**, **sp**, **LR**, **lr**, **PC** and **pc** are predefined by default. Names may also be defined for the registers used by the ARM Procedure Call Standard; see *Controlling syntax* on page 14. The syntax is:

symbol RN numeric-expression

FN defines the names of FPA floating point registers. The names **F0-F7** and **f0-f7** are predefined by default. The syntax is:

symbol FN numeric-expression

SN defines the names of VFP single-precision registers. The names **S0-S31** and **s0-s31** are predefined by default. The syntax is:

symbol SN numeric-expression«***.data-type***»

where *data-type* is any valid Advanced SIMD data type.

DN defines the names of VFP double-precision registers, 64-bit Advanced SIMD vectors, and Advanced SIMD scalars. The names **D0-D31** and **d0-d31** are predefined by default. The syntax is:

symbol DN numeric-expression«.element-type»[index]»»

where *element-type* is any valid Advanced SIMD data type, and *index*, if present, indicates that the symbol defines a scalar consisting of that index into the vector.

QN defines the names of 128-bit Advanced SIMD vectors. The names **Q0-Q15** and **q0-q15** are predefined by default. The syntax is:

symbol QN numeric-expression«.element-type»

where *element-type* is any valid Advanced SIMD data type.

You can omit any otherwise mandatory data type qualifiers on UAL opcodes, provided that all of the registers it references were declared using SN, DN or QN directives which included *element-type* qualifiers. This allows ObjAsm to deduce the appropriate instruction automatically.

CP gives a name to a coprocessor number, which must be within the range 0 to 15. The names **p0-p15** are predefined by default.

CN names a coprocessor register number; **c0-c15** are predefined by default. The syntax is:

symbol CP numeric-expression

symbol CN numeric-expression

Local and global variables – GBL, LCL and SET

While most symbols have fixed values determined during assembly, variables have values which may change as assembly proceeds. The assembler supports both global and local variables. The scope of global variables extends across the entire source file while that of local variables is restricted to a particular instantiation of a macro (see the chapter *Macros* on page 91). Variables must be declared before use with one of these directives.

GBLA Declares a global arithmetic variable. Values of arithmetic variables are 32-bit unsigned integers.

GBLL Declares a global logical variable

GBLS Declares a global string variable

LCLA Declares and initialises a local arithmetic variable (initial state zero)

LCLL Declares and initialises a local logical variable (initial state false)

LCLS Declares and initialises a local string variable (initial state null string)

The syntax of these directives is:

directive variable-name

The value of a variable can be altered using the relevant one of the following three directives:

SETA Sets the value of an arithmetic variable

SETL Sets the value of a logical variable

SETS Sets the value of a string variable

The syntax of these directives is:

variable-name directive expression

where *expression* evaluates to the value being assigned to the variable named.

(You can also declare and set the value of global variables at assembly time; see page 36.)

Variable substitution – \$

Once a variable has been declared its name cannot be used for any other purpose, and any attempt to do so will result in an error. However, if the \$ character is prefixed to the name, the variable's value will be substituted before the assembler checks the line's syntax. Logical and arithmetic variables are replaced by the result of performing a **:STR:** operation on them (see *Unary operators* on page 81); string variables are replaced by their value.

Aliases

Once a symbol has been defined, you can define another symbol to have the same value.

ALIAS *symbol, alias*

This defines symbol *alias* to the same value as *symbol*. *symbol* and *alias* can be EXPORTed with differing symbol attributes.

Built-in variables

ObjAsm provides a wide selection of built-in variables. They are:

{ARCHITECTURE}	String	The architecture in use. Architecture names are currently 1, 2, 2a, 3, 3G, 3M, 4xM, 4, 4TxM, 4T, 5xM, 5, 5TxM, 5T, 5TEXP, 5TE, 5TEJ, 6, 6K, 6T2, 6Z, 6-M, 6S-M, 7, 7-A, 7-R, 7-M, 7E-M.
{AREANAME}	String	Name of the current AREA.
{CODESIZE}	Arithmetic	Has the value 16 when working in Thumb mode, and the value 32 when working in ARM mode.
{CONFIG}	Arithmetic	Has the value 16 when working in Thumb mode. In ARM mode, the value depends on the APCS qualifiers: has the value 26 if /26bit is used and the value 32 if /32bit is used.
{CPU}	String	Name of the target CPU, or " Generic ARM " if only an architecture was selected.
{ENDIAN}	String	Has the value " big " if the assembler is in big-endian mode, and the value " little " if it is in little-endian mode.
{FALSE}	Logical	Logical constant false.
{FPU}	String	Name of the target FPU.
{INPUTFILE}	String	Name of the source file currently being processed.
{INTER}	Logical	{TRUE} when interworking is selected (i.e. APCS qualifier /interwork is used).
{LINENUM}	Arithmetic	Line number currently being processed.
{LINENUMUP}	Arithmetic	When in a macro, the line number from which the current macro was invoked, otherwise the same as {LINENUM}.
{LINENUMUPPER}	Arithmetic	When in a macro, the line number from which the outermost macro was invoked, otherwise the same as {LINENUM}.
{OBJASM_VERSION}	Arithmetic	The version of ObjAsm, multiplied by 100.
{OPT}	Arithmetic	Currently set listing option. The OPT directive can be used to save the current listing option, force a change in it or restore its original value.

{PC} or .	Arithmetic or program- relative	Current value of the program location counter.
{PCSTOREOFFSET}	Arithmetic	The offset added when storing PC to memory in the current CPU. An error is generated if only an architecture was selected.
{REENTRANT}	Logical	{TRUE} if APCS qualifier /reentrant (or one of its aliases) was used.
{ROPI}	Logical	{TRUE} if APCS qualifier /ropi (or one of its aliases) was used.
{RWPI}	Logical	{TRUE} if APCS qualifier /rwpi (or one of its aliases) was used.
{TARGET_ARCH_ <i>architecture</i> }	Logical	{TRUE} if specified architecture is currently selected. Valid values for <i>architecture</i> are: 1, 2, 2A, 3, 3G, 3M, 4XM, 4, 4TXM, 4T, 5XM, 5, 5TXM, 5T, 5TEXP, 5TE, 5TEJ, 6, 6K, 6T2, 6Z, 6_M, 6S_M, 7, 7_A, 7_R, 7_M, 7E_M
{TARGET_ARCH_ARM}	Arithmetic	Version number of the ARM instruction set supported by the current architecture, or 0 if only Thumb is supported.
{TARGET_ARCH_THUMB}	Arithmetic	Version number of the Thumb instruction set supported by the current architecture, or 0 if only ARM is supported.
{TARGET_FEATURE_CLZ}	Logical	{TRUE} if specified CPU supports the CLZ instruction - i.e. it supports ARM v5 and/or Thumb v4.
{TARGET_FEATURE_DIVIDE}	Logical	{TRUE} if specified CPU supports the SDIV and UDIV instructions - i.e. profile R or M of architecture 7.
{TARGET_FEATURE_DOUBLEWORD}	Logical	{TRUE} if specified CPU supports LDRD and related instructions - i.e. whether it has the P extension.

{TARGET_FEATURE_DSPMUL}	Logical	{TRUE} if specified CPU supports QADD and related instructions - i.e. whether it has the E extension.
{TARGET_FEATURE_EXTENSION_REGISTER_COUNT}	Arithmetic	How many double-precision floating point registers the specified VFP architecture provides, or 0 if VFP not selected.
{TARGET_FEATURE_MULTIPLY}	Logical	{TRUE} if specified CPU supports SMULL and related instructions - i.e. whether it has the M extension.
{TARGET_FEATURE_MULTIPROCESSING}	Logical	{TRUE} if specified CPU supports the ARMv7 multiprocessing extensions.
{TARGET_FEATURE_NEON}	Logical	{TRUE} if specified CPU provides any Advanced SIMD extension.
{TARGET_FEATURE_NEON_FP16}	Logical	{TRUE} if specified CPU provides the Advanced SIMD extension, including floating point support, including half-precision floating point instructions.
{TARGET_FEATURE_NEON_FP32}	Logical	{TRUE} if specified CPU provides the Advanced SIMD extension, including floating point support.
{TARGET_FEATURE_NEON_INTEGER}	Logical	{TRUE} if specified CPU provides any Advanced SIMD extension, including integer support.
{TARGET_FEATURE_UNALIGNED}	Logical	{TRUE} if specified CPU can perform unaligned memory accesses - i.e. architecture 6 or later, excluding v6-M and v6S-M.
{TARGET_FPU_FPA}	Logical	{TRUE} if a FPA-style FPU has been specified.

{TARGET_FPU_SOFTFPA}	Logical	{TRUE} if a FPA-endianness SoftFP calling standard is selected, and no hardware FP instructions are permitted.
{TARGET_FPU_SOFTFPA_FPA}	Logical	{TRUE} if a FPA-endianness SoftFP calling standard is selected, and FPA instructions are permitted.
{TARGET_FPU_SOFTFPA_VFP}	Logical	{TRUE} if a FPA-endianness SoftFP calling standard is selected, and VFP instructions are permitted.
{TARGET_FPU_SOFTVFP}	Logical	{TRUE} if a VFP-endianness SoftFP calling standard is selected, and no hardware FP instructions are permitted.
{TARGET_FPU_SOFTVFP_FPA}	Logical	{TRUE} if a VFP-endianness SoftFP calling standard is selected, and FPA instructions are permitted.
{TARGET_FPU_SOFTVFP_VFP}	Logical	{TRUE} if a VFP-endianness SoftFP calling standard is selected, and VFP instructions are permitted.
{TARGET_FPU_VFP}	Logical	{TRUE} if a VFP-style FPU has been specified.
{TARGET_FPU_VFPV2}	Logical	{TRUE} if specified FPU uses (precisely) version 2 of the VFP architecture.
{TARGET_FPU_VFPV3}	Logical	{TRUE} if specified FPU uses (precisely) version 3 of the VFP architecture.
{TARGET_FPU_VFPV4}	Logical	{TRUE} if specified FPU uses (precisely) version 4 of the VFP architecture.
{TARGET_PROFILE_A}	Logical	{TRUE} if specified CPU is of profile A of architecture 7.

{TARGET_PROFILE_M}	Logical	{TRUE} if specified CPU is of profile M of architecture 6 or 7.
{TARGET_PROFILE_R}	Logical	{TRUE} if specified CPU is of profile R of architecture 7.
{TRUE}	Logical	Logical constant true.
{UAL}	Logical	{TRUE} if ObjAsm is configured to expect UAL syntax instructions.
{VAR} or @	Arithmetic, register-relative or program-relative	Current value of the storage-area location counter.

It has never been possible to test if a given version of ObjAsm supports a given built-in variable, and if you attempt to use one that does not exist, you get an error. To facilitate conditional assembly based upon the available feature set, you can use the arithmetic variable **|objasm\$version|**, which is predefined by ObjAsm 4 or later, and has the same value as {OBJASM_VERSION}. For example:

```

MACRO
WhereAmI
IF :DEF: |objasm$version|
INFO 0, "In area \" :CC: {AREANAME} :CC: \"\"
ELSE
INFO 0, "In unknown area"
ENDIF
MEND

```

8

Expressions and operators

Expressions are combinations of simple values, unary and binary operators, and brackets. There is a strict order of precedence in their evaluation: expressions in brackets are evaluated first, then operators are applied in precedence order. Adjacent unary operators evaluate from right to left; binary operators of equal precedence are evaluated from left to right.

The assembler includes an extensive set of operators for use in expressions, many of which resemble their counterparts in high-level languages.

Single-character strings can be automatically converted to arithmetic expressions if the context demands it.

When intra-file Thumb code addresses are used in expressions, ObjAsm will set bit 0 of the address. This facilitates Thumb interworking using load instructions (from ARM v5) and ALU instructions (from ARM v7). Inter-file references to Thumb code addresses are fixed up by the Linker instead.

Unary operators

Unary operators have the highest precedence (bind most tightly) so are evaluated first. A unary operator precedes its operand, and adjacent operators are evaluated from right to left.

Operator	Usage	Explanation
!	!A	Logical complement of A
+	+A	Unary plus
-	-A	Unary negate. + and - can act on numeric, PC-relative and register-relative expressions.
?	?A	Number of bytes generated by line defining label A .

Operator	Usage	Explanation
BASE	:BASE:A	If A has no register offsets and no relocations, BASE produces an error and INDEX has no effect.
INDEX	:INDEX:A	If A has no register offsets and one or more relocations (e.g. it is a label from a non-BASED AREA), BASE returns 15 and INDEX has no effect. If A has a single, positive register offset (e.g.. it is a label from a BASED AREA, or it was defined using a FIELD directive where the preceding MAP directive referenced a register) and 0 or more relocations, BASE returns the number of the offset register and INDEX removes the register offset from the expression, leaving any relocations unaffected. Otherwise, both BASE and INDEX produce errors. BASE and INDEX are most likely to be of use within macros.
CC_ENCODING	:CC_ENCODING:A	Numeric value of an ARM instruction condition field (in bits 28-31) corresponding to the condition name string A
CHR	:CHR:A	ASCII string of A
DEF	:DEF:A	{TRUE} if A is defined, otherwise {FALSE}
LEN	:LEN:A	Length of string A
LNOT	:LNOT:A	Logical complement of A
LOWERCASE	:LOWERCASE:A	String A with each constituent character forced to lower-case.
NOT	:NOT:A	Bitwise complement of A
RCONST	:RCONST:A	A no-op in ObjAsm, and is provided for armasm compatibility. This is because ObjAsm still permits register names in expressions (they are automatically converted to the register number - this ability has been withdrawn in armasm).
REVERSE_CC	:REVERSE_CC:A	Opposite condition name string to that stated in string A .

Operator	Usage	Explanation
STR	:STR:A	Hexadecimal string of A . STR returns an eight-digit hexadecimal string corresponding to a numeric expression, or the string T or F if used on a logical expression.
UPPERCASE	:UPPERCASE:A	String A with each constituent character forced to upper-case.

Binary operators

Binary operators are written between the pair of sub-expressions on which they operate. Operators of equal precedence are evaluated in left to right order. The binary operators are presented below in groups of equal precedence, in decreasing precedence order.

Multiplicative operators

These are the binary operators which bind most tightly and have the highest precedence:

Operator	Usage	Explanation
%	A%B	A modulo B
*	A*B	Multiply
/	A/B	Divide
MOD	A:MOD:B	A modulo B

These operators act only on numeric expressions.

String manipulation operators

Operator	Usage	Explanation
CC	A:CC:B	B concatenated on to the end of A
LEFT	A:LEFT:B	The leftmost B characters of A
RIGHT	A:RIGHT:B	The rightmost B characters of A

In the two slicing operators **LEFT** and **RIGHT**, **A** must be a string and **B** must be a numeric expression.

Shift operators

Operator	Usage	Explanation
<<	A<<B	Shift A left B bits
>>	A>>B	Shift A right B bits
ROL	A:ROL:B	Rotate A left B bits
ROR	A:ROR:B	Rotate A right B bits
SHL	A:SHL:B	Shift A left B bits
SHR	A:SHR:B	Shift A right B bits

The shift operators act on numeric expressions, shifting or rotating the first operand by the amount specified by the second. Note that **SHR** and **>>** perform a logical shift and does not propagate the sign bit.

Addition and logical operators

Operator	Usage	Explanation
&	A&B	Bitwise AND of A and B
+	A+B	Add A to B
-	A-B	Subtract B from A
^	A^B	Bitwise Exclusive OR of A and B
	A B	Bitwise OR of A and B
AND	A:AND:B	Bitwise AND of A and B
EOR	A:EOR:B	Bitwise Exclusive OR of A and B
OR	A:OR:B	Bitwise OR of A and B

The bitwise operators act on numeric expressions. The operation is performed independently on each bit of the operands to produce the result.

Relational operators

Operator	Usage	Explanation
!=	A!=B	A not equal to B
/=	A/=B	A not equal to B
<	A<B	A less than B
<=	A<=B	A less than or equal to B
<>	A<>B	A not equal to B
=	A=B	A equal to B
==	A==B	A equal to B
>	A>B	A greater than B
><	A><B	A not equal to B
>=	A>=B	A greater than or equal to B

The relational operators act upon two operands of the same type to produce a logical value. Allowable types of operand are numeric, program-relative, register-relative, and strings. Strings are sorted using ASCII ordering. String **A** will be less than string **B** if it is either a leading substring of string **B**, or if the left-most character of **A** in which the two strings differ is less than the corresponding character in string **B**. Note that arithmetic values are unsigned, so the value of **0>-1** is **{FALSE}**.

Boolean operators

These are the weakest binding operators with the lowest precedence.

Operator	Usage	Explanation
&&	A&&B	Logical AND of A and B
 	A B	Logical OR of A and B
LAND	A:LAND:B	Logical AND of A and B
LOR	A:LOR:B	Logical OR of A and B
LEOR	A:LEOR:B	Logical Exclusive OR of A and B

The Boolean operators perform the standard logical operations on their operands, which should evaluate to **{TRUE}** or **{FALSE}**.

9

Conditional and repetitive assembly

This chapter describes the features available within the Assembler for constructing conditional assembly statements and conditional looping statements.

Conditional assembly

The `[` and `]` directives mark the start and finish of sections of the source file which are to be assembled only if certain conditions are true. The basic construction is `IF... THEN... ENDIF`; however, `ELSE` and `ELIF` are also supported, giving the full `IF... THEN... ELSE... ELIF... ENDIF` conditional assembly.

The start of the section is known as the `IF` directive:

```
[ logical_expression or IF logical_expression
```

This is the `ELIF` directive:

```
ELIF logical_expression
```

This is the `ELSE` directive:

```
| or ELSE
```

and this is the `ENDIF` directive:

```
] or ENDIF
```

A block which is being conditionally assembled can contain several `[|]` directives; that is, conditional assembly can be nested.

Simple use of the IF and ENDIF directives

You can use the IF and ENDIF directives (without the ELSE directive) like this:

```
[ logical_expression
.....
...code...
.....
]
```

The code will only be assembled if the logical expression is true; it will be skipped if the logical expression is false.

Simple use of the IF, ELSE and ENDIF directives

You can use three directives, thus:

```
[ logical_expression
.....
...first piece of code...
.....
|
.....
...second piece of code...
.....
]
```

If the logical expression is true, the first piece of code will be assembled and the second skipped. If the expression is false, the first piece of code will be skipped and the second assembled.

Simple use of the IF, ELIF, ELSE and ENDIF directives

Alternatively you can use all four directives, thus:

```
IF first_logical_expression
.....
...first piece of code...
.....
ELIF second_logical_expression
.....
...second piece of code...
.....
ELSE
.....
...third piece of code...
.....
ENDIF
```

If the first logical expression is true, the first piece of code will be assembled and the second and third skipped, irrespective of the truth of the second expression. If the first expression is false and the second is true, the first and third piece of code will be skipped and the second assembled. If both the first and second expressions

are false, the first and second piece of code will be skipped and the third assembled. If the third piece of code does nothing, you can choose to omit the ELSE directive.

Conditional assembly and the Terse listing option

Lines conditionally skipped by these directives are not listed unless ObjAsm is switched from its default terse mode. For desktop assembly, you must deselect **Terse listing** from ObjAsm's menu (see *Listings* on page 22); for command line usage, you must specify the **--no_terse** command line option (see page 31).

An example

An example of a notional data storage routine is given below. This routine can either use a disc or a tape data storage system. To assemble the code for tape operation, the programmer prepares the system by altering just one line of code, the label **SWITCH**.

```
DISC    *        0
TAPE    *        1
SWITCH  *        DISC
        ...code...
        [ SWITCH=TAPE
        ...tape interface code...
        ]
        [ SWITCH=DISC
        ...disc interface code...
        ]
        ...code continues...
```

or alternatively:

```
DISC    *        0
TAPE    *        1
SWITCH  *        DISC
        ...code...
        [ SWITCH=TAPE
        ...tape interface code...
        |
        ...disc interface code...
        ]
        ...code continues...
```

The IF construction can be used inside macro expansions as easily as it is used in the main program.

Repetitive assembly

It is often useful for program segments and macros to produce tables. To do this, they must be able to have a conditional looping statement. The Assembler has the WHILE... WEND construction. This produces an assembly time (not runtime) loop.

The syntax is:

WHILE *logical_expression*

to start the repetitive block, and:

WEND

to end it.

For example:

```
        GBLA    counter
counter SETA    100

        WHILE  counter >0
        DCD    &$counter
counter SETA    counter-1
        WEND
```

produces the same result as the following (but is shorter and less prone to typing errors):

```
        DCD    100
        DCD    99
        DCD    98
        DCD    97
        :
        DCD    2
        DCD    1
```

Since the test for the WHILE condition is made at the top of the loop, it is possible that the source within the loop will not generate any code at all.

Listing of conditionally skipped lines is as for conditional assembly.

10 Macros

Macros give you a means of placing a single instruction in your source which will be expanded at assembly time to several assembler instructions and directives, just as if you'd written those instructions and directives within the source at that point.

As an example, we will define a `TestAndBranch` instruction. This would normally take two ARM instructions. So we tell the Assembler, by means of a macro definition, that whenever it meets the `TestAndBranch` instruction, it is to insert the code we have given it in the macro definition. This is of course a convenience; we could just as easily write the relevant instructions out each time, but instead we let the Assembler do it for us.

The Assembler determines the destination of the branch with a macro parameter. This is a piece of information specified each time the macro is coded; the macro definition specifies how it is used. In the `TestAndBranch` example, we might also make the register to be tested a parameter, and even the condition to be tested for. Thus our macro definition might be:

```
MACRO
$label TestAndBranch$cc $dest,$reg      ; This is called the macro prototype
                                           ; statement
$label CMP      $reg,#0                 ; These two lines are the ones that
      B$cc      $dest                   ; will be substituted in the source.
      MEND                                     ; This says the macro definition is
                                           ; finished
```

A use of the macro might be:

```
Test    TestAndBranchNE NonZero,R0
      :
      :
      :
NonZero
```

The result, as far as the Assembler is concerned, is:

```
Test    CMP      R0,#0
      BNE      NonZero
      :
      :
      :
NonZero
```

Syntax

The fact that a macro is about to be defined is given by the directive **MACRO** in the instruction field:

MACRO

This is immediately followed by a macro prototype statement which takes the form:

<\$label> macroname<\$suffix> <\$parameter><,\$parameter>...

<\$label> if present, it is treated as an additional parameter.

<\$suffix> if present, it is treated as an additional parameter.

<\$parameter> Parameters are passed to the macro as strings and substituted before syntax analysis. Any number of them may be given.

The purpose of the macro prototype statement is to tell the Assembler the name of the macro being defined. The name of the macro is found in the opcode field of the macro prototype statement.

The macro prototype statement also tells the Assembler the names of the parameters, if any, of the macro. Parameters may occur in three places in the macro prototype statement. A single optional parameter may occur in the label field, shown as **\$label** above. This is normally used if the macro expansion is to contain a program label, and is merely an aid to clarity, as can be seen in the TestAndBranch example. An optional suffix to the macro name, shown as **\$suffix** above, forms another parameter. There is no special requirement that this is an ARM condition code, although this is what it will normally be used for. Any number of parameters, separated by commas, may occur in the operand field. All parameter names begin with the character **\$**, to distinguish them from ordinary program labels.

The macro prototype statement can also tell the Assembler the default values of any of the parameters. This is done by following the parameter name by an equals sign, and then giving the default value. If the default value is to begin or end with a space then it should be placed within quotes. For example:

```
$reg      = R0
$string  = " a string "
```

It is not possible to give a default value for the parameters in the label or suffix fields.

For example:

```

MACRO
$label MACRONAME $num,$string,$etc
.....
.....
$label ...lots of...
.....code....
=      $num
=      $string
=      "the price is $etc"
=      0
MEND

```

- **MACRONAME** is the name of this particular macro and **\$num**, **\$string** and **\$etc** are its parameters. Other macros may have many more parameters, or even none at all.
- The body of the macro follows after **MACRONAME**, with **\$label** being optional even if it was given in the macro prototype statement.
- **\$etc** will be substituted into the string **"the price is "** when the macro is used.
- The macro ends with **MEND**.

The macro is called by using its name and any missing parameters are indicated by commas, or may be omitted entirely if no more parameters are to follow. Thus, **MACRONAME** may be called in various ways:

```
MACRONAME      9,"disc",7
```

OR:

```
MACRONAME      9
```

OR:

```
MACRONAME      ,"disc",
```

Local variables

Local variables are similar to global variables, but may only be referenced within the macro expansion in which they were defined. They must be declared before they are used. The three types of local variable are arithmetic, logical and string. These are declared by:

Directive	Local variable type	Initial state
LCLA	Arithmetic	zero
LCLL	Logical	FALSE
LCLS	String	null string.

New values for local variables are assigned in precisely the same way as new variables for global variables: that is, using the directives **SETA**, **SETL** and **SETS**.

Syntax: *variable_name SETx expression*

Directive	Local variable type
SETA	Arithmetic
SETL	Logical
SETS	String

MEXIT directive

Normally, macro expansion terminates on encountering the **MEND** directive, at which point there must be no unclosed **WHILE/WEND** loops or pieces of conditional assembly. Early termination of a macro expansion can be forced by means of the **MEXIT** directive, and this may occur within **WHILE/WEND** loops and conditional assembly.

Default values

Macro parameters can be given default values at macro definition time, using the syntax:

\$parameter=default_value

In the example of the macro **MACRONAME** already used:

```

MACRO
$label MACRONAME $num,$string,$etc
.....
.....
$label ...lots of...
.....code....
=      $num
=      $string
=      "the price is $etc"
=      0
MEND
    
```

you could instead write ***\$num=10*** in the macro prototype statement. Then, when calling the macro, a vertical bar character '**|**' will cause the default value **10** to be used rather than the value ***\$num***. For example:

```
MACRONAME |,"disc",7
```

will be equivalent to:

```
MACRONAME 10,"disc",7
```

Note that this default is not used when the macro argument is omitted – the value is then empty.

Macro substitution method

Each line of a macro is scanned so it can be built up in stages before being passed to the syntax analyser. The first stage is to substitute macro parameters throughout the macro and then to consider the variables. If string variables, logical variables and arithmetic variables are prefixed by the **\$** symbol, they are replaced by a string equivalent. Normal syntax checking is performed upon the line after these substitutions have been performed.

An important exception to these values is that vertical bar characters (‘|’) prevent substitution from taking place in some circumstances. To be specific, if a line contains vertical bars, substitution will be turned off after this first vertical bar, on again after the second one, off again after the third, and so on. This allows the use of dollar characters in symbols and labels (see the section *Symbols* on page 44 for details).

In certain circumstances, it may be necessary to prefix a macro parameter or variable to a label. In order to ensure that the Assembler can recognise the macro parameter or variable, it can be terminated by a dot ‘.’ The dot will be removed during substitution.

For example:

```

MACRO
$T33  MACRONAME
      .....
      .....
$T33.L25...lots of...
      ....code....
MEND

```

If the dot had been omitted, the Assembler would not have related the **\$T33** part of the label to the macro statement and would have accepted **\$T33L25** as a label in its own right, which was not the intention.

Nesting macros

The body of a macro can contain a call to another macro; in other words, the expansion of one macro can contain references to macros. Macro invocation may be nested up to a depth of 255.

A division macro

As a final example, the following macro does an unsigned integer division:

```

; A macro to do unsigned integer division. It takes four parameters, each of
; which should be a register name:
;
; $Div: The macro places the quotient of the division in this register -
;       ie $Div := $Top DIV $Bot.
;       $Div may be omitted if only the remainder is wanted.
; $Top: The macro expects the dividend in this register on entry and places
;       the remainder in it on exit - ie $Top := $Top MOD $Bot.
; $Bot: The macro expects the divisor in this register on entry. It does not
;       alter this register.
; $Temp: The macro uses this register to hold intermediate results. Its initial
;        value is ignored and its final value is not useful.
;
; $Top, $Bot, $Temp and (if present) $Div must all be distinct registers.
; The macro does not check for division by zero; if there is a risk of this
; happening, it should be checked for outside the macro.

        MACRO
$Label  DivMod  $Div,$Top,$Bot,$Temp
        ASSERT $Top <> $Bot           ; Produce an error if the
        ASSERT $Top <> $Temp          ; registers supplied are
        ASSERT $Bot <> $Temp          ; not all different.
        [
        " $Div" /= ""
        ASSERT $Div <> $Top
        ASSERT $Div <> $Bot
        ASSERT $Div <> $Temp
        ]

$Label  MOV     $Temp,$Bot             ; Put the divisor in $Temp
        CMP     $Temp,$Top,LSR #1     ; Then double it until
90      MOVLS  $Temp,$Temp,LSL #1     ; 2 * $Temp > $Top.
        CMP     $Temp,$Top,LSR #1
        BLS    %b90
        [
        " $Div" /= ""
        MOV     $Div,#0              ; Initialise the quotient.
        ]

91      CMP     $Top,$Temp            ; Can we subtract $Temp?
        SUBCS  $Top,$Top,$Temp       ; If we can, do so.
        [
        " $Div /= ""
        ADC     $Div,$Div,$Div       ; Double $Div & add new bit
        ]
        MOV     $Temp,$Temp,LSR #1   ; Halve $Temp,
        CMP     $Temp,$Bot           ; and loop until we've gone
        BHS    %b91                  ; past the original divisor.
        MEND

```

The statement:

```
Divide DivMod R0,R5,R4,R2
```

would be expanded to:

```

    ASSERT R5 <> R4           ; Produce an error if the
    ASSERT R5 <> R2           ; registers supplied are
    ASSERT R4 <> R2           ; not all different
    ASSERT R0 <> R5
    ASSERT R0 <> R4
    ASSERT R0 <> R2
Divide MOV R2,R4             ; Put the divisor in R2.
      CMP R2,R5,LSR #1      ; Then double it until
90     MOVLS R2,R2,LSL #1   ; 2 * R2 > R5.
      CMP R2,R5,LSR #1
      BLS %b90
91     MOV R0,#0           ; Initialise the quotient.
      CMP R5,R2           ; Can we subtract R2?
      SUBCS R5,R5,R2      ; If we can, do so.
      ADC R0,R0,R0        ; Double R0 & add new bit.
      MOV R2,R2,LSR #1   ; Halve R2,
      CMP R2,R4           ; and loop until we've gone
      BHS %b91           ; past the original divisor.

```

Similarly, the statement:

```
DivMod ,R6,R7,R8
```

would be expanded to:

```

    ASSERT R6 <> R7           ; Produce an error if the
    ASSERT R6 <> R8           ; registers supplied are
    ASSERT R7 <> R8           ; not all different.
    MOV R8,R7             ; Put the divisor in R8.
    CMP R8,R6,LSR #1      ; Then double it until
90     MOVLS R8,R8,LSL #1   ; 2 * R8 > R6.
    CMP R8,R6,LSR #1
    BLS %b90
91     CMP R6,R8           ; Can we subtract R8?
    SUBCS R6,R6,R8      ; If we can, do so.
    MOV R8,R8,LSR #1   ; Halve R8,
    CMP R8,R7           ; and loop until we've gone
    BHS %b91           ; past the original divisor.

```

Note:

- Conditional assembly is used to reduce the size of the assembled code (and increase its speed) in the case where only the remainder is wanted.
- Local labels are used to avoid multiply defined labels if **DivMod** is used more than once in the assembler source.
- The letter '**b**' is used in the local label references (indicating that the Assembler should search backwards for the corresponding local labels) to ensure that the correct local labels are found.

Part 3 – Developing software for RISC OS

This chapter discusses ways of manipulating the Processor Status Register (PSR) which maintain compatibility across a range of processors including the ARM2, ARM3, ARM6, ARM7, Strong ARM, XScale and Cortex-A8 processors.

To just set and clear NZCV flags you can use macros which do the right thing for the different processor types. To actually preserve flags, you will probably be forced to use MRS and MSR instructions. These are NOPs on pre-ARM 6 ARMs, so it is possible to write code sequences which work on all processors from ARM2 to Cortex-A8.

If writing code that is only for 26-bit modes or only 32-bit modes then the simple macros supplied in Libraries.Hdr may be used. Set the logical switches **No26bitCode** or **No32bitCode** as required, then **GET Hdr:CPU.Generic26** and **Hdr:CPU.Generic32**. The **No26bitCode** switch means don't rely on 26-bit instructions (e.g. TEQP and LDM ^) – the code will work on 32-bit systems. The switch **No32bitCode** switch means don't rely on 32-bit instructions (e.g. MSR and MRS) - the code will work on RISC OS 3.1. Setting both to {TRUE} is too much for the macros to cope with so to achieve compatibility across the whole range of processors from you will have to use run-time code as shown below.

The recommended general-purpose code to check whether you're in a 26-bit mode is:

```
TEQ    R0, R0          ; Sets Z (can be omitted if not in User mode)
TEQ    PC, PC         ; EQ if in a 32-bit mode, NE if 26-bit
```

Here is an example of calling a SWI from an IRQ routine:

```
TEQ    PC, PC          ; EQ if in 32-bit mode
MRSEQ  R8, CPSR        ; Save CPSR in R8 (32-bit mode case)
MOVNE  R8, PC          ; Save CPSR in R8 (26-bit mode case)
ORR    R9, R8, #3      ; IRQ26->SVC26, IRQ32->SVC32
MSREQ  CPSR_c, R9      ; Switch to SVC32 (32-bit mode case)
TEQNEP R9, #0          ; Switch to SVC26 (26-bit mode case)
NOP                                          ; NOP to avoid problems on ARM2
STR    R14, [R13, #-4]!; faster than STMFD on some new processors
SWI    XOS_AddCallBack
LDR    R14, [R13], #4  ; ditto
TEQ    PC, PC          ; EQ if in a 32-bit mode
MSREQ  CPSR_c, R8      ; Restore CPSR (32-bit mode case)
TEQNEP R8, #0          ; Restore CPSR (26-bit mode case)
NOP
```

Note it would theoretically be possible to arrange for the TEQP to occur before the MSR and hence have the MSR be the required NOP for the ARM 2, but this would require knowledge of what the TEQP will do to the Z flag, and it appears that the Strong ARM has a bug invoked when TEQP is followed by MSR, even if they're not both executed.

The complexity of the above example occurs because of the need to support pre-ARM 6 processors that don't have the MRS and MSR instruction (i.e. RISC OS 3.1 machines). If RISC OS 3.1 support is not required, it reduces to:

```
MRS    R8, CPSR
ORR    R9, R8, #3      ; IRQ26->SVC26, IRQ32->SVC32
MSR    CPSR_c, R9
STR    R14, [R13, #-4]! ; faster than STMFd on some new processors
SWI    XOS_AddCallBack
LDR    R14, [R13], #4  ; ditto
MSR    CPSR_c, R8
```

This is possible because the MRS and MSR instructions are available on ARM6 and ARM7 processors even when running in 26-bit mode.

Sometimes you may be forced to manipulate the SPSR registers. Beware with interrupt code which will corrupt SPSR_svc if it calls a SWI. Existing interrupt handlers know to preserve R14_svc before calling a SWI, but not SPSR_svc. Hence you MUST disable interrupts around SPSR manipulations; the SPSR is not suitable as a general mechanism for PSR restoration on function return.

12 Writing relocatable modules in assembler

Relocatable modules are the basic building blocks of RISC OS and the means by which RISC OS can be extended by a user.

The relocatable module system provides mechanisms suitable for

- providing device drivers
- extending the set of RISC OS *commands
- providing shared services to applications (eg the shared C library)
- implementing 'terminate and stay resident' (TSR) applications.

All these projects require code either to be more persistent than standard RISC OS applications or to be used by more than one application, hence resident in the address space of more than one application. If your program does not have these requirements it is not recommended to put it in modules, as relocatable modules are more persistent consumers of system resources than applications, and are also more difficult to debug.

This chapter is not intended to provide a complete set of the technical details you need to know to construct any relocatable module. For more information on such details, see the RISC OS 3 *Programmer's Reference Manual*. The points covered here are intended to provide help for constructing relocatable modules specifically in assembly language.

For more details of memory management in relocatable modules, you should again see the RISC OS 3 *Programmer's Reference Manual*.

Unlike the construction of relocatable modules in high level languages, no tools are provided to generate substantial standard portions of code. This means that you have to construct the module header table, workspace routines, etc. yourself.

Note that some of the relocatable module entry points are called in SVC mode. Such routines may use SWIs implemented by other parts of RISC OS, but unlike being in user mode, SWIs corrupt R14, so this must be stored away. Floating point instructions can be used from SVC mode in RISC OS 4 and later.

Assembler directives

ObjAsm can be used to assemble a module from a set of source files, a link step being required to join the output object files to form the usable module. The separation of routines into separately assembled files has several advantages.

It can be a good idea to construct a module with the module header and the small routines/data associated with it in one source file, to be linked with the code forming the body of the module.

Such a module header file must be linked so that it is placed first in the module binary. To do this it should contain an **AREA** directive at its head such as:

```
AREA |!!!Module$$Header|, CODE, READONLY
```

Areas are sorted by type and name; a name beginning with '!' is placed before an alphabetic name, so the above can be used to ensure first placing.

The module header source needs to contain **IMPORT** directives making available any symbols referenced in the module body. In addition, the initialisation routine should call **__RelocCode**, a routine added by the linker which relocates any absolute references to symbols when the module is initialised. If the module header source contains the initialisation routine, it must use the **IMPORT** directive to make **__RelocCode** available.

The module header must be preceded by the **ENTRY** directive:

```
ENTRY
```

```
Module_BaseAddr
```

```

DCD    RM_Start      -Module_BaseAddr
DCD    RM_Init       -Module_BaseAddr
DCD    RM_Die        -Module_BaseAddr
DCD    RM_Service    -Module_BaseAddr
DCD    RM_Title      -Module_BaseAddr
DCD    RM_HelpStr    -Module_BaseAddr
DCD    RM_HC_Table   -Module_BaseAddr
DCD    RM_SWIChunk   -Module_BaseAddr
DCD    RM_SWIHandler -Module_BaseAddr
DCD    RM_SWINames   -Module_BaseAddr
DCD    RM_SWIDecode  -Module_BaseAddr
DCD    RM_Messages   -Module_BaseAddr
DCD    RM_Flags      -Module_BaseAddr

```

Example

This product is supplied with the source for an example relocatable module that provides an extra soft screen mode: Mode 63. This has to be done via service call handling, and to be useful must be persistent, so providing a typical use of relocatable modules.

There are two source files held in **AcornC_C++.Examples.AsmModule.s**:

- The **ModeExHdr** file produces the module header, and may be useful for you to copy and edit to form headers for your own modules.
- The other file, **ModeExBody**, is the source for the main module body.

To build the module, use ObjAsm to assemble the source. Then link the resultant object files using Link, remembering first to set the **Module** option on its Setup dialogue box.

The module is specific to VIDC1 and VIDC1a, and so will not work on Acorn computers that are fitted with later versions of VIDC – such as the Risc PC.

13 Interworking assembler with C

Interworking assembly language and C – writing programs with both assembly language and C parts – requires using both ObjAsm and C/C++.

Interworking assembly language and C allows you to construct top quality RISC OS applications. Using this technique you can take advantage of many of the strong points of both languages. Writing most of the bulk of your application in C allows you to take advantage of the portability of C, the maintainability of a high level language, and the power of the C libraries and language. Writing critical portions of code in assembler allows you to take advantage of all the speed of the Archimedes and all the features of the machine (eg the complete floating point instruction set).

The key to interworking C and assembler is writing assembly language procedures that obey the ARM Procedure Call Standard (APCS). This is a contract between two procedures, one calling the other. The called procedure needs to know which ARM and floating point registers it can freely change without restoring them before returning, and the caller needs to know which registers it can rely on not being corrupted over a procedure call. Additionally both procedures need to know which registers contain input arguments and return arguments, and the arrangement of the stack has to follow a pattern that debuggers, etc. can understand. For the specification of the APCS, see the appendix *ARM procedure call standard* on page 263 of the accompanying *Desktop Tools* guide.

Examples

The following examples are provided to demonstrate how to write programs combining assembly language and C.

PrintLib

The directory **AcornC_C++.Examples.PrintLib.s** contains three source files from which you can build a library: **PrintStr**, **PrintHex** and **PrintDble**. These are the assembly language sources for three screen printing routines: **print_string**, **print_hex** and **print_double**. These respectively print null terminated strings, integers in hexadecimal, and double precision floating point numbers in scientific format.

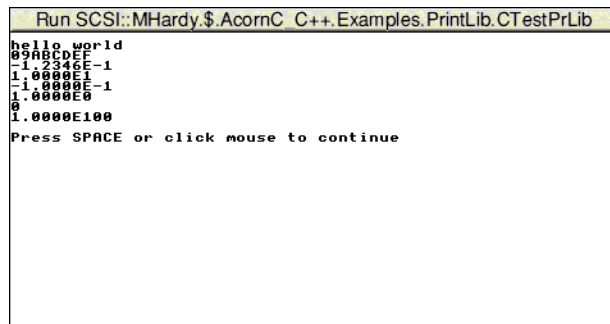
Each routine is written to obey the APCS, so it can be called from assembler, C, or any other high level language obeying the APCS. The sources for PrintLib illustrate several aspects of the APCS, such as the distinction between leaf and non-leaf procedures, and how floating point arguments are passed into a procedure.

Compiling the CTestPrLib example

To show you that you can call the routines in PrintLib from C, we've supplied a small C program in **AcornC_C++.Examples.PrintLib.c.CTestPrLib**. To build this example, you must:

- 1 Build the PrintLib library; you'll find instructions for this in the section *Assembler example* on page 134 of the *Desktop Tools* guide.
- 2 Start CC if you've not already got it loaded.
- 3 Drag the **CTestPrLib** file to the CC icon, which will display its **Setup** dialogue box with **CTestPrLib** already entered as the source to compile.
- 4 Add the full pathname of the PrintLib library to the list of **Libraries** on the Setup menu.
- 5 Click on Run to compile and link the program.
- 6 Save the program to disc.

To run the program, double click on its icon in the directory display to which you saved it. A standard RISC OS command line output window appears containing text printed by the assembly language library routines as a result of arguments passed from C:



```
Run SCSI::MHardy$.AcornC_C++.Examples.PrintLib.CTestPrLib
hello world
00ABCDEF
-1.2345E-1
1.0000E1
-1.0000E-1
1.0000E0
0
1.0000E100
Press SPACE or click mouse to continue
```

Compiling and linking CTestPrLib in separate stages

If you prefer, you can instead use the **Compile only** option of CC to compile **CTestPrLib** to an object file.

You can then use Link to link this object file with the libraries it uses. As well as the PrintLib library, it also uses the C library, so you must link three files: the object code for **CTestPrLib**, the library built from the PrintLib source, and the C library stubs held in **AcornC_C++.Libraries.clib.o.stubs**.

(In the above section *Compiling the CTestPrLib example*, the C library stubs were linked in because they were already in the Setup menu's default list of **Libraries**.)

CStatics

The directory **AcornC_C++.Examples.CStatics** gives an example of accessing C static variables from both assembler and C source code. The example builds to form a relocatable module providing a single * Command: ***CStatics**.

The files in the directory are as follows:

- **c.CInit** is the C source code. It declares two variables: **extern int var1**, which is provided by and initialised to 0 in **s.AsmInit** (see below), and **int var2**, which it initialises as 0. It prints the values of the two variables. It then calls the routine **Asm_Change_Vars** provided by **s.AsmInit** (see below), which changes the values of the two variables. Finally it prints the new values.
- **cmhg.Header** is the CMHG description file for the module. **hdr.CVars** is an assembler source file that contains a series of macros used by **s.AsmInit**. You will find these useful if you too ever need to share static data between assembler and C.
- **MakeFile** is the make file for the CStatics module.
- **o** is an empty directory used to hold the object files created when making the CStatics module.
- **s.AsmInit** is the assembler source code. It initialises the variable **var1** to 0 and exports it; it also imports the variable **var2**. It also provides an APCS conformant routine **Asm_Change_Vars** which adds 10 to **var1** and subtracts 10 from **var2**. All this code makes heavy use of the macros in **hdr.CVars**.

To build the CStatics module, simply double click on the MakeFile.

When Make has completed, you can see the example in use. Load the resultant **CStatics** module by double clicking on it, then type **CStatics** at the command line. You will get this output:

```
var1 = 0
var2 = 0
var1 = 10
var1 = -10
```

If you repeat the ***CStatics** command you will see the variables change again:

```
var1 = 10
var2 = -10
var1 = 20
var1 = -20
```

and so on, every time you repeat the command.

Part 4 – Appendices

Appendix A: Changes to the assembler

This release of ObjAsm replaces the version previously supplied as part of the Castle/Acorn C/C++ Development Environment, the last major release of which was in 2002. The changes from that version are considerable:

- Apart from deprecated switches, all multi-character switches can now (and are preferred to) be specified in GNU style: preceded by a double dash, and with '=' between the switch and its value (if applicable). -noesc, -noterse and -nowarn gain an underscore to become --no_esc, --no_terse and --no_warn to match armasm.
- Support for architectures 3G, 6, 6K, 6T2, 6Z, 6-M, 6S-M, 7, 7-A, 7-R, 7-M, 7E-M and 7-A.security
- Support for CPUs ARM600, ARM610, ARM700, ARM704, ARM710, ARM720T, SA-110, SA-1100, SA-1110, ARM910T, ARM920T, ARM922T, ARM926EJ-S, ARM7TM-S, ARM7TDM, ARM7TDMI-S, ARM710T, ARM740T, ARM7EJ-S, ARM810, ARM925T, ARM940T, ARM946E-S, ARM966E-S, ARM968E-S, ARM10E, ARM10EJ-S, ARM1020T, ARM1020E, ARM1022E, ARM1026EJ-S, 88FRxxx(.no_hw_divide), ARM1136J-S(-rev1), ARM1136JF-S(-rev1), ARM1156T2(F)-S, ARM1176JZ(F)-S, MPCore(.no_vfp), Cortex-M0, Cortex-M1(.no_os_extension), Cortex-M3(-rev0), SC300, Cortex-M4(.fp), Cortex-R4(F), Cortex-A5(.vfp.neon), Cortex-A8(.no_neon), Cortex-A9(.no_neon(.no_vfp)), QSP(.no_neon(.no_vfp))
- --cpu=list can be used to print a list of supported CPUs
- -m is no longer the short form of -module, but has the same meaning as in armasm, viz send dynamic dependency information to stdout
- --debug is a new alias for -g
- New command-line switches (documented elsewhere): --arm, --arm_only, --checkreglist, --cpreproc, --cpreproc_opts, --device, --fpu, --keep, --regnames, --no_code_gen
- The APCS version number (3) can now be omitted
- Additional APCS qualifiers are now accepted: /fp, /nonreent(rant), /softfp, /hardfp, /(no)fpr(egargs), /fpe2, /fpe3, /fpa, /vfp, /26, /32, /(no)pic, /(no)ropi, /(no)pid, /(no)rwpi
- All built-in variable names are now matched case-insensitively.

-
- The following new variables are added: {AREANAME}, {CPU}, {FPU}, {INPUTFILE}, {LINENUM}, {LINENUMUP}, {LINENUMUPPER}, {OBJASM_VERSION}, {PCSTOREOFFSET}, {ROPI}, {RWPI}, {TARGET_ARCH_architecture}, {TARGET_ARCH_ARM}, {TARGET_ARCH_THUMB}, {TARGET_FEATURE_CLZ}, {TARGET_FEATURE_DIVIDE}, {TARGET_FEATURE_DOUBLEWORD}, {TARGET_FEATURE_DSPMUL}, {TARGET_FEATURE_EXTENSION_REGISTER_COUNT}, {TARGET_FEATURE_MULTIPLY}, {TARGET_FEATURE_MULTIPROCESSING}, {TARGET_FEATURE_NEON}, {TARGET_FEATURE_NEON_FP16}, {TARGET_FEATURE_NEON_FP32}, {TARGET_FEATURE_NEON_INTEGER}, {TARGET_FEATURE_UNALIGNED}, {TARGET_FPU_FPA}, {TARGET_FPU_SOFTFPA}, {TARGET_FPU_SOFTFPA_FPA}, {TARGET_FPU_SOFTFPA_VFP}, {TARGET_FPU_SOFTVFP}, {TARGET_FPU_SOFTVFP_FPA}, {TARGET_FPU_SOFTVFP_VFP}, {TARGET_FPU_VFP}, {TARGET_FPU_VFPV2}, {TARGET_FPU_VFPV3}, {TARGET_FPU_VFPV4} {TARGET_PROFILE_A}, {TARGET_PROFILE_R}, {TARGET_PROFILE_M}, {UAL}, lobjasm\$versionl
 - New area attributes: NOSWSTACKCHECK, VFP, CODEALIGN
 - New pseudo-instructions: IT, MOV32, UND, VMOV2
 - Added literal-loading forms of LDRSB, LDRH, LDRSH, LDRD
 - MOV, MVN, ADR and literal forms of LDR etc can output MOVW
 - ADRL can output MOV32
 - Shifts (other than LSL) by zero are permitted as pseudo-instructions
 - New directives: ALIAS, ARM, DCDU, DCFDU, DCFH, DCFHU, DCFSU, DCQ, DCQU, DCWU, ELIF, EXPORTAS, FILL, REQUIRE
 - ALIGN can now pad with nonzero values or with NOPs
 - DCB now accepts values between -128 and -1
 - DCI takes optional .N or .W suffix
 - EQU takes a new second parameter
 - EQU and MAP can reference labels in other areas of the source file
 - EXPORT, IMPORT, KEEP and LEAF default qualifiers are set by --apcs /softfp/hardfp/fpr/nofpr
 - EXPORT with no arguments
 - EXPORT supports additional qualifiers: NOFPREGARGS, SOFTFP, HARDFP, CODE, NONLEAF, NOUSESSB, THUMB, ARM
 - EXTERN now only outputs a symbol if it is referenced in source code
 - IMPORT [WEAK] can be combined with other qualifiers inside the {}

- IMPORT supports additional qualifiers: NOFPREGARGS, SOFTFP, HARDFP, DATA, CODE, NOWEAK, READONLY, READWRITE, BASED
- New third parameter to INFO and !
- INFO 0,*string*,0 now prints filename and line number
- MACRO names can now take an optional suffix parameter
- The following directives can now define local labels: =, &, %, DCB, DCD, DCDO, DCDU, DCFD, DCFDU, DCFH, DCFHU, DCFS, DCFSU, DCI, DCQ, DCQU, DCW, DCWU, FILL, SPACE
- Word-type operators (unary and binary) are now matched case-insensitively
- New unary operators: !, LOWERCASE, UPPERCASE, REVERSE_CC, CC_ENCODING, RCONST
- BASE and INDEX have been extended to be useful with external symbols and BASED areas
- New binary operators: %, ==, !=, &, &&, ||, ^
- The following operators and their aliases can now act on external symbols and symbols from other areas within the same file, where appropriate for the context:
=, <>, <, <=, >=, >, +, -
- You can now use the inequality operators on logical expressions.
- New ARM instructions:
 - XScale: MAR, MIA, MIABB, MIABT, MIAPH, MIATB, MIATT, MRA
 - ARMv6: CPS, LDREX, MCRR2, MRRC2, PKHBT, PKHTB, QADD16, QADD8, QADDSUBX, QASX, QSAX, QSUB16, QSUB8, QSUBADDX, REV, REV16, REVSH, RFE, SADD16, SADD8, SADDSUBX, SASX, SEL, SETEND, SHADD16, SHADD8, SHADDSUBX, SHASX, SHSAX, SHSUB16, SHSUB8, SHSUBADDX, SMLAD(X), SMLALD(X), SMLSD(X), SMLSLD(X), SMMLA(R), SMMLS(R), SMMUL(R), SMUAD(X), SMUSD(X), SRS, SSAT(16), SSAX, SSUB16, SSUB8, SSUBADDX, STREX, SXT(A)B(16), SXT(A)H, UADD16, UADD8, UADDSUBX, UASX, UHADD16, UHADD8, UHADDSUBX, UHASX, UHSAX, UHSUB16, UHSUB8, UHSUBADDX, UMAAL, UQADD16, UQADD8, UQADDSUBX, UQASX, UQSAX, UQSUB16, UQSUB8, UQSUBADDX, USAD(A)8, USAT(16), USAX, USUB16, USUB8, USUBADDX, UXT(A)B(16), UXT(A)H
 - ARMv6K: CLREX, LDREXB, LDREXD, LDREXH, STREXB, STREXD, STREXH
 - ARMv6Z: SMC, SMI
 - ARMv6T2: BFC, BFI, LDRHT, LDRSBT, LDRSHT, MLS, MOVt, MOVW, RBIT, SBFX, STRHT, UBFX

-
- ARMv6K or ARMv6T2: DBG, SEV, WFE, WFI, YIELD
 - ARMv7: DMB, DSB, ISB, PLI
 - ARMv7 multiprocessing extension: PLDW
 - VFPv1/v2: FABSD, FABSS, FADDD, FADDS, FCMPD, FCMPED, FCMPE, FCMPED, FCMPEZD, FCMPEZS, FCMPD, FCMPZD, FCMPZS, FCPYD, FCPYS, FCVTDS, FCVTSD, FDIVD, FDIVS, FLDD, FLDMDBD, FLDMDBS, FLDMDBX, FLDMEAD, FLDMEAS, FLDMEAX, FLDMFDD, FLDMFDS, FLDMFDX, FLDMIAD, FLDMIAS, FLDMIAX, FLDS, FMACD, FMACS, FMDHR, FMDLR, FMDRR, FMRDH, FMRDL, FMRRD, FMRRS, FMRS, FMRX, FMSCD, FMSCS, FMSR, FMSRR, FMSTAT, FMULD, FMULS, FMXR, FNEGD, FNEGS, FNMACD, FNMACS, FNMSCD, FNMSCS, FNMULD, FNMULS, FSITOD, FSITOS, FSQRTD, FSQRTS, FSTD, FSTMDBD, FSTMDBS, FSTMDBX, FSTMEAD, FSTMEAS, FSTMEAX, FSTMFD, FSTMFD, FSTMFDX, FSTMIAD, FSTMIAS, FSTMIAX, FSTS, FSUBD, FSUBS, FTOSID, FTOSIS, FTOSIZD, FTOSIZS, FTOUID, FTOUIS, FTOUIZD, FTOUIZS, FUITOD, FUITOS, VABS, VADD, VCMD, VCMPE, VCVT, VCVTR, VDIV, VLDM, VLDMDB, VLDMEA, VLDMFD, VLDMIA, VLDR, VMLA, VMLS, VMOV, VMRS, VMSR, VMUL, VNEG, VNMLA, VNMLS, VNMUL, VPOP, VPUSH, VSQRT, VSTM, VSTMDB, VSTMEA, VSTMFD, VSTMIA, VSTR, VSUB
 - VFPv3: FCONSTD, FCONSTS, FSHTOD, FSHTOS, FSLTOD, FSLTOS, FTOSH, FTOSH, FTOSLD, FTOSLS, FTOUHD, FTOUHS, FTOULD, FTOULS, FUHTOD, FUHTOS, FULTOD, FULTOS
 - VFP half-precision extension: VCVTB, VCVTT
 - VFPv4: VFMA, VFMS, VFNMA, VFNMS
 - Advanced SIMD: VABA, VABAL, VABD, VABDL, VACGE, VACGT, VACLE, VACLT, VADDHN, VADDL, VADDW, VAND, VBIC, VBIF, VBIT, VBSL, VCEQ, VCGE, VCGT, VCLE, VCLS, VCLT, VCLZ, VCNT, VDUP, VEOR, VEXT, VHADD, VHSUB, VLD1, VLD2, VLD3, VLD4, VMAX, VMIN, VMLAL, VMLSL, VMOVL, VMOVN, VMULL, VMVN, VORN, VORR, VPADAL, VPADD, VPADDL, VPMAX, VPMIN, VQABS, VQADD, VQDMLAL, VQDMLSL, VQDMULH, VQDMULL, VQMOVN, VQMOVUN, VQNEG, VQRDMULH, VQRSHL, VQRSHRN, VQRSHRUN, VQSHL, VQSHLU, VQSHRN, VQSHRUN, VQSUB, VRADDHN, VRECPE, VRECPS, VREV16, VREV32, VREV64, VRHADD, VRSHL, VRSHR, VRSHRN, VRSQRT, VRSQRTS, VRSRA, VRSUBHN, VSHL, VSHLL, VSHR, VSHRN, VSLI, VSRA, VSRI, VST1, VST2, VST3, VST4, VSUBHN, VSUBL, VSUBW, VSWP, VTBL, VTBX, VTRN, VTST, VUZP, VZIP
 - Thumb-only instructions are errors: CBNZ, CBZ, CHKA, ENTERX, HB(L)(P), LEAVEX, ORN, SDIV, TBB, TBH, UDIV

- New Thumb instructions (all ARMv6 / Thumbv3): CPS, CPY, REV, REV16, REVSH, SETEND, SXTB, SXTH, UXTB, UXTH
- There was some extremely old code in objasm which was designed to handle source code written for the Unix 'as' assembler. However, this was undocumented and had been broken for a long time. Even when it was functional, it only supported 'a.out' format object files, not AOF. To simplify objasm, this feature has been removed (note that it has also been removed from armasm).
- Layout improvements have been made in the output generated by --xref
- Floating-point constants can now be expressed in new formats: "0f_" or "0d_" prefixes, or using non-decimal bases
- Labels on a line by themselves in code areas, when inserted into the symbol table by a KEEP or EXPORT directive, used to be marked as code unless overridden by the EXPORT [DATA] qualifier. In line with armasm, such labels now default to data if they follow a directive that outputs data.
- Removed the restriction that only one BASED area with a given base register was permitted per source file.
- Expressions can now use imported symbols and symbols from other areas at arbitrary positions, including multiple references to these types of symbols, even if this means emitting multiple relocations for the same word (which was never previously possible in objasm).
- ADR and ADRL can now be used on symbols from based areas.
- LDR can now be used on absolute addresses, and outputs a PC-relative-to-absolute relocation (though this is likely to be of limited use in practice).
- LDF, STF, LFM, SFM, LDC or STC can now be used on external or inter-area symbols.
- When Thumb code addresses are stored as words, bit 0 is now set.
- Unconditional instructions can now be given an AL condition code
- NV instructions generate an error from ARMv5 onwards
- Several additional warnings of unpredictable behaviour on old instructions added.

UAL differences

The assembler now supports UAL syntax. The differences this implies are:

- # is always optional

-
- # is optionally accepted before any field that accepts an immediate constant but not an address label: the second immediate in rotator-form ALU instructions, coprocessor instructions, BKPT, etc
 - Condition codes come after other modifier characters, except for TEQP etc and FPA opcodes
 - Optional '.W' width specifier after condition code but before VFP type specifier (if any)
 - LDM, RFE, SRS and STM default to IA if addressing mode is unspecified
 - LDRD/STRD quote both Rt registers
 - Shifting MOV instructions have simplified aliases, e.g. RRXEQ R0,R1 (in all of which, Rd is optional)
 - Rd is optional in ADC, ADD, AND, BIC, EOR, MUL, ORR, PKHBT, PKHTB, QADD, (Q|S|SH|U|UH|UQ)ADD8, (Q|S|SH|U|UH|UQ)ADD16, QDADD, QDSUB, QSUB, (Q|S|SH|U|UH|UQ)SUB8, (Q|S|SH|U|UH|UQ)SUB16, RSB, RSC, SBC, SEL, SMMUL(R), SMUAD(X), SMULxy, SMULWy, SMUSD(X), SUB, (SIU)XT(A)B(16), (SIU)XT(A)H, USAD8
 - (Q|S|SH|U|UH|UQ)ADDSUBX is renamed (Q|S|SH|U|UH|UQ)ASX and Rd is optional
 - (Q|S|SH|U|UH|UQ)SUBADDX is renamed (Q|S|SH|U|UH|UQ)SAX and Rd is optional
 - SWI is renamed SVC and SMI is renamed SMC
 - POP and PUSH are aliases for LDMFD sp! and STMFD sp!, and are automatically converted to equivalent LDR or STR instructions for single-register lists
 - CPY is an an alias for MOV unshifted-register
 - NEG(S) is an alias for RSB(S) from #0
 - ASL #0, ASR #0, LSL #0, LSR #0, ROR #0 are all valid and assemble to LSL #0
 - In MRS and MSR, old-style PSR suffixes (_all, _flg, _ctl) are no longer permitted
 - In MRS, APSR is an alias for CPSR
 - In MSR, APSR_{nzcvcq}{g} is an alias for CPSR_{f}{s}, and APSR is an alias for CPSR_f
 - In MSR, CPSR and SPSR are no longer permitted without a suffix
 - In MRC{2}, APSR_nzcv is used instead of PC
 - SRS must quote SP (and the optional '!' is after SP, not after mode)
 - For anything later than ARMv6, NOP assembles to a dedicated hint instruction rather than MOV r0,r0

Appendix B: Differences from RVDS

ObjAsm shares a common heritage with the assembler included with ARM's SDT, ADS and RVDS toolchains, which is commonly known as `armasm`. Development of ObjAsm has progressed in parallel with `armasm`, but some differences have inevitably arisen. Here is a non-exhaustive list of the differences between ObjAsm 4 and the `armasm` from RVDS 4.1.

- The following switches are not supported by `objasm`: `--brief_diagnostics`, `--compatible`, `--depend_format`, `--device_opt`, `--diag_error`, `--diag_remark`, `--diag_style`, `--diag_suppress`, `--diag_warning`, `--dllexport_all`, `--dwarf2`, `--dwarf3`, `--exceptions_unwind`, `--fpmode`, `--library_type`, `--licretry`, `--list=(default filename variant)`, `--md`, `--memaccess`, `--(no-)exceptions`, `--no_exceptions_unwind`, `--(no-)execstack`, `--no_hide_all`, `--no_project`, `--(no_)reduce_paths`, `--no_regs`, `--(no_)unaligned_access`, `--project`, `--reinitialize_workdir`, `--report-if-not-wysiwyg`, `--show_cmdline`, `--split_ldm`, `--thumb`, `--thumbx`, `--unsafe`, `--untyped_local_labels`, `--version_number`, `--vsn`, `--workdir`
- The following switches are not supported by `armasm`: `objasm`'s deprecated options, `--object`, `--absolute`, `--desktop`, `--no_cache`, `--throwback`, `--uppercase`
- `objasm` doesn't support the following `--cpu` values (alternative names are available): `MPCoreNoVFP` `Cortex-A8NoNEON`
- `armasm` doesn't support the following `--cpu` values: `1`, `2`, `2a`, `3`, `3G`, `3M`, `4xM`, `4TxM`, `5xM`, `5`, `5TxM`, `5TEXP`, `ARM1`, `ARM2`, `ARM3`, `ARM6`, `ARM600`, `ARM610`, `ARM7`, `ARM700`, `ARM704`, `ARM710`, `ARM710a`, `ARM710C`, `ARM7M`, `ARM7DM`, `ARM8`, `StrongARM`, `StrongARM1`, `SA-1110`, `ARM10TDMI`, `ARM10E`, `ARM10EJ-S`, `ARM1020T`
- `armasm` doesn't support the following `--fpu` values: `SoftFPA`, `SoftFPA+anything`, `FPE2`, `FPE3`, `FPA11`, `FPA11`, `VFPv1-SP`, `VFPv1`, `VFPv2-SP`, `VFPv3-SP`, `VFPv3-SP_FP16`
- `armasm` doesn't support the following `--device` values: `ARM250`, `ARM7100`, `ARM7500`, `ARM7500FE`
- `armasm` defaults to `--cpu=ARM7TDMI` while `objasm` has not changed its default from `--cpu=3`

-
- If you select a device/cpu which doesn't have hardware floating point, armasm effectively defaults to `--fpu=SoftVFP --apcs=/softfp/vfp`, whereas, for compatibility with earlier versions, objasm defaults to `--fpu=FPE2 --apcs=3/hardfp/nofpregargs/fpa/fpe2`
 - objasm doesn't support the following APCS qualifiers: `/(no)fpic`
 - armasm doesn't support `--apcs 3`, nor any of the following APCS qualifiers: `/26(bit)`, `/32(bit)`, `/(non)reent(rant)`, `/fpe2`, `/fpe3`, `/(no)fpr(egargs)`, `/swst(ackcheck)`, `/nosw(stackcheck)`, `/(no)fp`, `/fpa`, `/vfp`
 - Variables defined in objasm but not in armasm are `{OBJASM_VERSION}`, `{REENTRANT}`, `{TARGET_ARCH_architecture}` for architectures not supported by armasm, `{TARGET_FPU_FPA}`, `{TARGET_FPU_SOFTFPA}`, `{TARGET_FPU_SOFTFPA_FPA}`, `{TARGET_FPU_SOFTFPA_VFP}`, `{TARGET_FPU_SOFTVFP_FPA}`, `{UAL}`, `lobjasm$version!`
 - Variables defined in armasm but not objasm are `{ARMASM_VERSION}`, `{COMMANDLINE}`, `{FPIC}`, `lads$version!`
 - armasm doesn't support the following AREA attributes: `ABS`, `PIC`, `A32bit`, `REENTRANT`, `FP3`, `BASED`, `HALFWORD`, `NOSWSTACKCHECK`, `VFP`
 - objasm doesn't support the following AREA attributes: `ASSOC`, `COMGROUP`, `FINI_ARRAY`, `GROUP`, `INIT_ARRAY`, `LINKORDER`, `MERGE`, `NOALLOC`, `PREINIT_ARRAY`, `SECFLAGS`, `SECTYPE`, `STRINGS`
 - For integer literal access, armasm only supports `LDR=` (not even `LDRB=`)
 - armasm permits relocations in `DCWU` and `DCDU` directives but objasm can't because the linker and C library don't support unaligned relocations
 - armasm has lost `LEAF`, `STRONG` and `ORG` directives, and has not acquired `DCFH` or `DCFHU`
 - objasm has not adopted the following armasm directives: `THUMB`, `THUMBX`, `REQUIRE8`, `PRESERVE8`, `RELOC`, `COMMON`, `FRAME`, `FUNCTION`, `PROC`, `ENDFUNC`, `ENDP`, `ATTR`
 - objasm does not support the following `IMPORT` attributes: `SIZE`, `ELFTYPE` or the ELF symbol visibility attributes
 - armasm does not support the following `IMPORT` attributes: `(NO)FPREGARGS`, `SOFTFP`, `HARDFP`, `NOWEAK`, `READWRITE`, `READONLY`, `BASED`
 - objasm does not support the following `EXPORT` attributes: `WEAK`, `SIZE`, `ELFTYPE` or the ELF symbol visibility attributes
 - armasm does not support the following `EXPORT` attributes: `(NO)FPREGARGS`, `SOFTFP`, `HARDFP`, `(NON)LEAF`, `(NO)USESSB`
 - armasm doesn't implement the unary `!` operator
 - armasm doesn't support `:INDEX:` on based or external symbols

- objasm automatically converts register symbols to arithmetic constants when they are used in an expression. armasm faults the use of register symbols in expressions except in conjunction with the :RCONST: operator.
- armasm does not support automatic narrowing or widening of "0f_" or "0d_" style floating point constants. It also interprets floating point constants starting "0" or "0x" like "0f_" or "0d_" values rather than as C99-style hexadecimal floating point. If you want to ensure that a hexadecimal integer is not interpreted by armasm in this way, you can add a ".", an exponent or a leading "+" or "-", which are all faulted by armasm. It also does not support floating point numbers using base 2 to base 9.
- objasm accepts half-precision floating point constants in VMOV, VMOV2 and VLDR (literal) instructions.
- armasm does not round floating point constants in VMOV instructions.
- armasm does not support ADR or ADRL on external symbols or on constants (except ADRL on a constant, which it incorrectly assembles as an offset from the current area).
- The only directive that armasm can usefully use on based symbols is DCDO - all the others generate incorrect relocations.
- armasm has dropped support for DCDO on a register-relative symbol which was defined using the # directive.
- armasm can't do PC-relative-to-constant relocations for LDR instructions.
- armasm doesn't support the following pseudo-instructions:
 - V<R>SRA *d,m,#0*
 - VSRI *d,m,#0*
 - VQSHL.*Usize.Ssize d,m,#imm*
 - VQMOVN.*Usize.Ssize d,m*
 - VQ<R>SHRN.*Usize.Ssize d,m,#imm*

Appendix C: Error messages

This appendix lists most of the common error messages that you may get when using the assembler, and gives an explanation for each one of the circumstances that may provoke the error.

Fatal errors

- **A label was found which was in no AREA**
An AREA directive must precede any label definition. This can happen if you accidentally try to assemble a file which does not contain an assembler program.
- **Structure mismatch**
A file included by a GET directive ended before a necessary ENDIF or WEND directive was found.

Other fatal errors may indicate a bug in ObjAsm, and you are encouraged to report them.

Errors

- **'\' should not be used to split strings**
If you need to do this, use paired quotes on each line and join the substrings using a :CC: directive.
- **ADRL can't be used with PC**
The destination register of an ADRL opcode cannot be PC.
- **Area directive missing**
An attempt has been made to generate code or data before the first AREA directive.
- **Area name missing**
The name for the area has been omitted from an AREA directive.
- **Assertion failed**
The argument to an ASSERT directive evaluated to {FALSE}
- **Bad absolute symbol qualifier**
Did not recognise the second parameter to a * directive.
- **Bad alias name**
The wording of the first parameter following the ALIAS directive is syntactically not a name.

-
- **Bad alias symbol type**
The symbol identified by the first parameter to an ALIAS directive is unsuitable for being aliased.
 - **Bad alignment boundary**
An alignment has been given which is not a power of two.
 - **Bad alignment pad size**
An unsupported size of padding value has been specified in an ALIGN directive (must be 1, 2 or 4).
 - **Bad area attribute or alignment**
Unknown attribute or alignment not in the range 2-12.
 - **Bad based number**
A digit has been given in a based number which is not less than the base, for example: 7_8.
 - **Bad condition code**
The parameter to an IT instruction, or a string operated on by :CC_ENCODING: or :REVERSE_CC:, was not a valid conditional execution suffix.
 - **Bad exported name**
The wording following the EXPORT, EXPORTAS, KEEP or LEAF directive is syntactically not a name.
 - **Bad exported symbol type**
The specified symbol is not suitable for being exported.
 - **Bad expression type**
For example, a number was given when a boolean expression was expected. Also produced if the expression requires relocations which are not allowed in the present context.
 - **Bad floating point constant**
The only allowed floating point immediate constants for FPA are 0, 1, 2, 3, 5, 10 and 0.5. They must be written in exactly these forms. VFP and Advanced SIMD instructions allow any valid syntax for floating point immediate constants, but the range of valid values varies from instruction to instruction.
 - **Bad fill value size**
An unsupported size of padding value has been specified in a FILL directive (must be 1, 2 or 4).
 - **Bad GET or INCLUDE**
The specified file was not found.
 - **Bad global name**
An incorrect character appears in the global variable name.

- **Bad hexadecimal number**
The & or 0x introducing a hexadecimal number is not followed by a valid hexadecimal digit.
- **Bad imported name**
The wording following the IMPORT, REQUIRE or STRONG directive is syntactically not a name.
- **Bad local label number**
A local label number must have a leading number (conventionally, but not necessarily, in the range 0-99).
- **Bad local name**
An incorrect character appears in the local variable name.
- **Bad macro name**
An incorrect character appears in the macro name in a macro definition.
- **Bad macro parameter default value**
For example, the default value has mismatched quotes.
- **Bad operand type**
An operator has been given one or more operands of a type it does not support. For example, a logical value was supplied where a string was required.
- **Bad operator**
The name between colons is not an operator name.
- **Bad or unknown attribute**
The only attribute allowed after the square brackets of an IMPORT directive is WEAK.
- **Bad PSR designator**
Expected CPSR, SPSR or a recognised PSR bitfield name.
- **Bad register list symbol**
An expression used as a register set definition (e.g. in LDM or STM) was not understood or of the wrong type.
- **Bad register name symbol**
A register name is wrong.
- **Bad register range**
A register range from a higher to a lower register has been given; for example, R4-R2 has been typed.
- **Bad rotator**
The rotator value supplied must be even and in the range 0-30.
- **Bad shift name**
Syntax error in shift name.

-
- **Bad string escape sequence**
A C style escape character sequence (beginning with '\') within a string was incorrect.
 - **Bad symbol**
Syntax error in a symbol name.
 - **Bad symbol type**
This will occur after a # or * directive and means that the symbol being defined has already been assumed to be of a type which cannot be defined in this way.
 - **B/BL to unaligned destination**
Attempt to branch to ARM code at a non-word-aligned address, using a B or BL instruction from ARM code, or a BLX instruction from Thumb code. There is no equivalent check for Thumb code because bit 0 is often used to indicate that this is a Thumb address.
 - **Branch offset out of range**
The destination of a branch is not within addressable space.
 - **Code generated in data area**
An opcode has been found in an area which is not a code area.
 - **Conflicting element indexes in list**
If you pass a list of Advanced SIMD scalars to $VLDn$ or $VSTn$ instructions, the element index must be identical for each scalar.
 - **Coprocessor number out of range**
The coprocessor number in a CP directive must be in the range 0-15.
 - **Coprocessor operation out of range**
The operation fields in generic coprocessor instructions must be in the range 0-7 or 0-15.
 - **Coprocessor register number out of range**
The coprocessor register number in a CN directive must be in the range 0-15.
 - **CPSR/SPSR_flg and CPSR/SPSR_ctl are illegal in MRS**
This operation can only be performed on a complete PSR.
 - **Data transfer offset out of range**
The immediate value in a data transfer opcode has limited range, and the range varies from instruction to instruction. See ARM ARM for details.
 - **Decimal overflow**
The number exceeds 32 bits (or 64 bits in a 64-bit expression).
 - **Division by zero**
Could not evaluate the results of a division or modulus operator.
 - **Encoding not available**
The specified instruction cannot be encoded using the specified instruction set and/or instruction width.

- **End of input file**
The END directive was not found.
- **Entry address already set**
This is the second or subsequent ENTRY directive.
- **Error in macro parameters**
The macro parameters do not match the prototype statement in some way.
- **Error on code file**
An error occurred while writing the output file.
- **Even-numbered register required**
In ARM code, Rt in LDRD and STRD instructions must be even-numbered.
- **Expected *required-type* data type for destination**
An invalid combination of Advanced SIMD data types has been specified for this instruction.
- **Expected *required-type* data type for first source**
An invalid combination of Advanced SIMD data types has been specified for this instruction.
- **Expected *required-type* data type for second source**
An invalid combination of Advanced SIMD data types has been specified for this instruction.
- **Expected 128-bit register symbol**
Only a Q register is permitted here.
- **Expected 32-bit ARM or 64-bit register or scalar symbol**
Only an R or D register or a scalar is permitted here.
- **Expected 32-bit ARM or VFP, 64-bit or 128-bit register or scalar symbol**
Only an R, S, D or Q register or a scalar is permitted here.
- **Expected 32-bit ARM or VFP or 64-bit register or scalar symbol**
Only an R, S or D register or a scalar is permitted here.
- **Expected 32-bit VFP, 64-bit or 128-bit register symbol**
Only an S, D or Q register is permitted here.
- **Expected 32-bit VFP or 64-bit register symbol**
Only an S or D register is permitted here.
- **Expected 32-bit VFP register symbol**
Only an S register is permitted here.
- **Expected 64-bit or 128-bit register or scalar symbol**
Only a D or Q register or a scalar is permitted here.

-
- **Expected 64-bit or 128-bit register symbol**
Only a D or Q register is permitted here.
 - **Expected 64-bit register or scalar symbol**
Only a D register or a scalar is permitted here.
 - **Expected 64-bit register symbol**
Only a D register is permitted here.
 - **Expected constant expression**
The expression was a string or boolean, or had a register or relocation component.
 - **Expected constant or address expression**
The expression was a string or a boolean, or had a register or relocation component other than a relocation to the current AREA.
 - **Expected string expression**
The expression was a number or boolean.
 - **Expected string or constant expression**
The expression was a boolean, or had a register or relocation component.
 - **Floating point number not found**
Missing or incorrect syntax for a floating point number.
 - **Floating point overflow**
A number was given which was too large to express as a half-precision floating point number. When a number is too large to express as a single or double-precision floating point number, it is silently converted to infinity.
 - **Floating point register number out of range**
FPA registers range from F0-F7.
 - **Global name already exists**
This name has already been used other than as a global variable of the specified type.
 - **Hexadecimal overflow**
The number exceeds 32 bits (or 64 bits in a 64-bit expression).
 - **Illegal combination of code and zero initialised**
An object file area cannot be declared both to be code and zero initialised data.
 - **Illegal label parameter start in macro prototype**
The label parameter must consist of a \$ followed by a valid symbol name
 - **Illegal line start should be blank**
A label has been found at the start of a line with a directive which cannot be labelled.

- **Illegal parameter in macro prototype**
The part of a macro parameter following the \$ must be a valid symbol name.
- **Illegal parameter start in macro prototype**
Macro parameters must begin with a \$.
- **Illegal shift for this instruction**
The instruction used does not support all barrel shift types.
- **Immediate value out of range**
Specified immediate value cannot be expressed by this instruction.
- **Imported name already exists**
The name has already been defined or used for something else.
- **Incorrect routine name**
The optional name following a branch to a local label or on a local label definition does not match the routine's name.
- **Instruction cannot be conditional in ARM instruction set**
This generally means the instruction is encoded in the 'NV' part of the instruction set, so there is no way to express the conditions under which it should be executed. Usually, this instruction can be conditionally executed if you use the Thumb instruction set instead.
- **Invalid data alignment for this combination of instruction, register list and element size**
An attempt has been made to specify an unsupported alignment after the @ symbol in a VLD n or VST n instruction.
- **Invalid line start**
A line may only start with a letter character (the first letter of a label), a digit (the first character of a local label), a semi-colon or a space.
- **Invalid parameter separator in macro prototype**
Use a comma between macro parameters.
- **Invalid register for instruction, did you mean APSR_nzcv?**
In UAL mode, MRC targeting R15 must use the name APSR_nzcv instead.
- **Invalid register for this instruction**
You cannot use the specified register here. Usually this is because the bitfield used to encode that register number is also used to identify the instruction type.
- **Invalid register for Thumb instruction**
Many Thumb instructions are restricted to using R0-R7.

-
- **Invalid register list for this instruction**
An attempt has been made to specify a combination of registers in a register list which is not supported by the current instruction Refer to the ARM ARM for supported lists.
 - **Invalid scalar for this instruction**
Advanced SIMD multiply instructions are unable to access the entire pool of scalars. The scalars available depend upon the element size.
 - **Label missing from line start**
The absence of a label where one is required; for example, in the * directive.
 - **LDRD destination register cannot be used as offset**
The effect of such an instruction is unpredictable.
 - **Line too long**
Try splitting the line using \ continuation characters.
 - **Literal pool too distant**
The instructions which load literals from literal pools have limited reach. Insert additional LTORG directives.
 - **Local label not permitted for this directive**
Local labels are only appropriate for directives that emit data.
 - **Local name already exists**
A local name has been defined more than once.
 - **Locals not allowed outside macros**
A local variable has been defined in the main body of the source file.
 - **Macro already exists**
A macro can only be defined once.
 - **Macro definitions cannot be nested**
A macro cannot be used to define another macro.
 - **Macro definition too big**
Macros are currently limited to a size of 4K.
 - **MEND not allowed within conditionals**
A MEND has been found amongst IF/ELIF/ELSE/ENDIF or WHILE/WEND directives.
 - **Missing at symbol**
An @ is absent.
 - **Missing close bracket**
A missing close bracket or too many opening brackets.
 - **Missing close curly bracket**
A } is absent.

- **Missing close quote**
No closing quote at the end of a string constant.
- **Missing close square bracket**
A] is absent.
- **Missing comma**
Syntax error due to missing comma.
- **Missing endianness option**
The SETEND instruction must be followed by BE or LE
- **Missing exclamation mark**
A required ! is absent.
- **Missing hash**
The hash (#) preceding an immediate value has been forgotten in pre-UAL syntax.
- **Missing open bracket**
A missing open bracket or too many closing brackets.
- **Missing open curly bracket**
An { is absent.
- **Missing open square bracket**
A [is absent.
- **Missing system register name**
The name of a VFP / Advanced SIMD system register is required.
- **MOV32 can't be used with PC**
The destination register of an MOV32 opcode cannot be PC.
- **Multiply or incompatibly defined symbol**
A symbol has been defined more than once.
- **No current macro expansion**
A MEND or MEXIT has been encountered but there is no corresponding MACRO.
- **Non-zero data within uninitialised area**
All data in a NOINIT AREA must have value 0.
- **No pre-declaration of substituted symbol**
Attempt to substitute the value of a variable which has not been defined yet.
- **Numeric overflow**
A based number exceeds 32 bits (or 64 bits in a 64-bit expression).
- **NV condition not permitted for targeted CPU**
From ARMv5 onwards, the bit pattern which used to express the NV condition code decodes to different or undefined instructions.

-
- **{PCSTOREOFFSET} is not defined when assembling for an architecture**
Use a specific CPU or device name to enable this built-in variable.
 - **Register list element sizes must match if no data type specified on instruction**
Either choose register symbols which were declared using the same element size, or use a data size qualifier on the opcode to override them.
 - **Register list must be in increasing register number order**
The --checkreglist option has detected a non-standard register list.
 - **Register name fp used in APCS /nofp mode**
Either use a different register name, or use the /fp option.
 - **Register occurs multiply in register list**
Any given register can only be used once in each list of registers.
 - **Registers must be contiguous**
This instruction requires that there be no gaps between the specified registers.
 - **Registers must be contiguous in ARM code**
The destination registers for LDRD and the source registers for STRD must be consecutive register numbers, except in Thumb code.
 - **Registers must match**
This instruction requires that the same register is specified twice.
 - **Register symbol already defined**
A register symbol has been defined as a different type of symbol, or with a different register number.
 - **Register value out of range**
Register values must be in the range 0-15, except FPA registers (0-7), VFP registers (0-31) and 64-bit Advanced SIMD vectors (0-31).
 - **Scalar index out of range**
Scalar indexes must be in the range 0-7 for 8-bit scalars, 0-3 for 16-bit scalars, or 0-1 for 32-bit scalars.
 - **Shift option out of range**
The range permitted is 0-31 or 0-32 depending on the shift type.
 - **Specified condition is not consistent with previous IT**
Correct either the condition field or the preceding IT instruction.
 - **Specified destination data type not allowed**
An invalid Advanced SIMD data type has been specified for this instruction.
 - **Specified source data type not allowed**
An invalid Advanced SIMD data type has been specified for this instruction.

- **String overflow**
Concatenation has produced a string of more than 512 characters.
- **String too short for operation**
An attempt has been made to manipulate a string using :LEFT: or :RIGHT: which has insufficient characters in it.
- **Structure mismatch**
IF/ELIF/ELSE/ENDIF must be in the correct order, and must be fully nested within any WHILE/WEND directives, and vice versa.
- **Structure stack overflow**
The level of nested IF blocks, WHILE loops and GET directives is limited to 256.
- **Structure stack underflow**
ELIF, ELSE, ENDIF or WEND without a preceding IF or WHILE.
- **Substituted line too long**
During variable and macro parameter substitution the line length has exceeded 4096 characters.
- **Symbol missing**
An attempt has been made to apply a ? or :DEF: operator, but the symbol was omitted or the name found was not recognised as a symbol. Or, an attempt to rename a symbol using an ALIAS or EXPORTAS directive was made, but the new symbol name was omitted or not recognised.
- **Syntax error following directive**
An operand has been provided to a directive which cannot take one, for example: the 'l' directive.
- **Syntax error following label**
A label can only be followed by spaces, a semi-colon or the end-of-line symbol.
- **Syntax error following local label definition**
A space, comment, or end-of-line did not immediately follow the local label.
- **Thumb code generation disabled**
Use of Thumb code when --arm_only switch has been used.
- **Too few data types specified on instruction**
Add more data type specifiers to the opcode, or remove all of them and use only register symbols which were declared with data types.
- **Too late to ban floating point instructions**
At least one floating point instruction has already been emitted.
- **Too late to change output format**
AOF and AOUT directives can only be used once, and only before any symbols have been defined or any code or data has been output.

-
- **Too late to define symbol as register list**
A register list was defined for a symbol already used for another purpose.
 - **Too late to set origin now**
The ORG must be set before the Assembler generates code.
 - **Too many actual parameters**
A macro call is trying to pass too many parameters.
 - **Too many data types specified on instruction**
Remove one or more data type specifier from the opcode.
 - **Translate not allowed in pre-indexed form**
The T flag cannot be specified in pre-indexed forms of LDR and STR.
 - **Unable to open output file**
Parent directory doesn't exist, exists as a directory, exists but access is denied, etc
 - **Undefined exported symbol**
The symbol mentioned in an EXPORT, KEEP or LEAF directive must also be defined somewhere in the source file.
 - **Undefined symbol**
A symbol used in an expression is not defined anywhere in the source file. Or, a symbol in an ALIAS, EXPORTAS or REQUIRE directive is not defined before the directive.
 - **Unexpected characters at end of line**
The line is syntactically complete, but more information is present. The semi-colon prefixing comments may have been omitted.
 - **Unexpected operand**
An operand has been found where a binary operator was expected.
 - **Unexpected operator**
A non-unary operator has been found where an operand was expected.
 - **Unexpected unary operator**
A unary operator has been found where a binary operator was expected.
 - **Unknown opcode**
A name in the opcode field has been found which is not an opcode, a directive, nor a macro.
 - **Unknown operand**
An unrecognised built-in variable has been used.
 - **Unknown or wrong type of global/local symbol**
Type mismatch, for example, attempting to set or reset the value of a local or global symbol as logical, where it is a string variable.

- **Unknown shift name**
Not one of the six legal shift mnemonics.
- **Unmatched conditional or macro**
END or LNK directive found before the necessary ENDIF/WEND directives have been found.
- **Unrecognised endianness option**
The SETEND instruction must be followed by BE or LE.
- **Unrecognised flags**
The flags that can be affected by a CPS instruction are A, I and F.
- **Unrecognised #line syntax**
ObjAsm uses #line directives inserted by the C preprocessor to identify the original file that a line came from. This indicates the #line directive is not of a supported format.
- **Unrecognised system register name**
The name given for a VFP / Advanced SIMD system register is not known to ObjAsm.
- **Unspecified endianness for DCFD or DCFDU**
ObjAsm does not know how to format double-precision floating point data if you use --fpu=none.
- **Useless instruction (PC can't be written back)**
Writeback to PC was used on an LDC or STC instruction.
- **Use of banked R8-R14 after forced user-mode LDM**
This has unpredictable effects on some CPUs.
- **Writeback to base not available with user mode transfer**
Adjusting the base register must be done in a separate instruction for LDM[^] and STM[^].

Warnings

- **'#' not seen before constant expression**
The hash character is optional in UAL syntax, but its omission may indicate an unintentional error.
- **'\' at end of comment**
A comment cannot be made to continue onto the next line in this way.
- **AOF symbol attribute not recognised**
An unknown attribute name was used in an IMPORT or EXPORT directive.
- **ARM not supported on targeted CPU**
The specified CPU only accepts Thumb instructions.

-
- **Deprecated form of PSR field specifier used (use `_cxsf`)**
Names like `CPSR_cxsf` are preferred to ones like `CPSR_all`. This is elevated to an error in UAL mode.
 - **Deprecated instruction (LDM with LR and PC in register list)**
 - **Deprecated instruction (LDM with SP in register list)**
 - **Deprecated instruction (LDM with writeback and base in register list; base register is left with unknown value)**
 - **Deprecated instruction (STM with SP or PC in register list)**
 - **Deprecated instruction (STM with writeback and base is first in register list)**
 - **Deprecated instruction (STM with writeback and base not first in register list; value stored from base register has unknown value)**
 - **Directive found within IT block**
Condition code checking may give incorrect results if the directive outputs data.
 - **Faking declaration of area `AREA |$$$$$$|`**
AREA directive was missing.
 - **Floating point out of range for IEEE half-precision, using alternative format**
VFP alternative format half-precision floating point allows a maximum exponent of 2^{16} , as opposed to IEEE half-precision where the maximum exponent is 2^{15} . This permits larger normalised numbers to be expressed, at the cost of not being able to express infinities or NaNs.
 - **Inexact floating point constant**
The floating point immediate constant provided to a `VMOV` instruction is not one of the architecturally supported values, but `ObjAsm` has been able to substitute an approximation.
 - **Instruction not supported on targeted CPU**
You have used an instruction which has no hardware support on the current CPU.
 - **LDM or STM of single register is probably slower than LDR or STR**
You may wish to change your code to use the equivalent `LDR` or `STR` instruction.

- **Macro ignores label parameter**
When a macro was invoked, it was given a label parameter, even though its definition does not accept one.
- **Macro ignores suffix parameter**
When a macro was invoked, it was given a suffix parameter, even though its definition does not accept one.
- **Missing END directive at end of file**
Files are required to finish with an END directive.
- **ORG base forced to word boundary**
ORG addresses must be word-aligned.
- **Pre-UAL syntax in UAL ARM code**
May indicate an unintentional syntax error.
- **Register bank wrap**
A group of registers that wrap around the end of the register bank has been specified on an instruction for which this results in unpredictable behaviour.
- **Register not supported on targeted FPU**
A double precision register has been referenced when the chosen variant of the VFP only supports single precision registers, or a register in the range D16-D31 has been referenced when the chosen variant of the VFP only supports 16 double precision registers.
- **Reserved instruction (using NV condition)**
These instructions are not available in ARMv5 and later. Use NOP instructions instead.
- **Specifying a PSR field specifier is deprecated (use CPSR or SPSR)**
MRS instructions apply to the whole of a PSR, not to a field within it. This is elevated to an error in UAL mode.
- **SWP is deprecated, use LDREX/STREX instead**
LDREX/CLREX is a better choice on CPUs that support those instructions.
- **Thumb not supported on targeted CPU**
The specified CPU only accepts ARM instructions.
- **UAL syntax in pre-UAL ARM code**
May indicate an unintentional syntax error.
- **Undefined effect (PC-relative SWP)**
- **Undefined effect (Rd = Rm in MUL/MLA instruction)**
- **Undefined effect (use of PC/PSR)**
- **Unpredictable instruction (invalid processor mode)**

-
- Unpredictable instruction (LDM with writeback and base in register list)
 - Unpredictable instruction (move immediate to PSR updates do-not-modify bits or sets write-as-zero bits)
 - Unpredictable instruction (PC-relative exclusive access)
 - Unpredictable instruction (PC used as an operand)
 - Unpredictable instruction (PC + writeback)
 - Unpredictable instruction (RdLo = RdHi)
 - Unpredictable instruction (Rd must differ from other registers)
 - Unpredictable instruction (Rd = PC)
 - Unpredictable instruction (Rm = RdLo or RdHi)
 - Unpredictable instruction (Rm = Rn with writeback)
 - Unpredictable instruction (source or destination same as written-back base)
 - Unpredictable instruction (SP used as an operand in Thumb code)
 - Unpredictable instruction (SWP to or from base register)
 - Unpredictable instruction (transfer of more than 16 64-bit registers)
 - Unpredictable instruction (unimplemented accumulator)
 - Use of banked R8-R14 after in-line mode change
This has unpredictable effects on some CPUs.
 - Use of CPY between low registers is unpredictable before ARMv6
Before ARMv6, you must use MOV(S) in Thumb code, even though it means setting the flags.

Appendix D: Example assembler fragments

The following example assembly language fragments show ways in which the basic ARM instructions can combine to give efficient code. None of the techniques illustrated save a great deal of execution time (although they all save some), mostly they just save code.

Note that, when optimising code for execution speed, consideration to different hardware bases should be given. Some changes which optimise speed on one machine may slow the code on another. An example is unrolling loops (eg divide loops) which speeds execution on an ARM2, but can slow execution on an ARM3, which has a cache.

Using the conditional instructions

Using conditionals for logical OR

```
CMP    Rn,#p           ; IF Rn=p OR Rm=q THEN GOTO Label
BEQ    Label
CMP    Rm,#q
BEQ    Label
```

can be replaced by:

```
CMP    Rn,#p
CMPNE  Rm,#q           ; If condition not satisfied try
BEQ    Label           ; another test.
```

Absolute value

```
TEQ    Rn,#0           ; Test sign
RSBMI  Rn,Rn,#0       ; and 2's complement if necessary.
```

Combining discrete and range tests

```
TEQ    Rc,#127        ; discrete test
CMPNE  Rc,#"-1"       ; range test
MOVLS  Rc,#"."        ; IF Rc<#" " OR Rc=CHR$127 THEN Rc:="."
```

Division and remainder

```

; Enter with dividend in Ra, divisor in Rb.
; Divisor must not be zero.
      MOV     Rd,Rb                ; Put the divisor in Rd.
      CMP     Rd,Ra,LSR #1        ; Then double it until
Div1  MOVLS   Rd,Rd,LSL #1        ; 2 * Rd > divisor.
      CMP     Rd,Ra,LSR #1
      BLS     Div1
      MOV     Rc,#0                ; Initialise the quotient
Div2  CMP     Ra,Rd                ; Can we subtract Rd?
      SUBCS   Ra,Ra,Rd            ; If we can, do so
      ADC     Rc,Rc,Rc            ; Double quotient and add new bit
      MOV     Rd,Rd,LSR #1        ; Halve Rd.
      CMP     Rd,Rb                ; And loop until we've gone
      BHS     Div2                ; past the original divisor,
; Now Ra holds remainder, Rb holds original divisor,
; Rc holds quotient and Rd holds junk.

```

Pseudo-random binary sequence generator

It is often necessary to generate (pseudo-) random numbers, and the most efficient algorithms are based on shift generators with a feedback rather like a cyclic redundancy check generator. Unfortunately, the sequence of a 32 bit generator needs more than one feedback tap to be maximal length (that is, $2^{32}-1$ cycles before repetition). A 33 bit shift generator with taps at bits 20 and 33 is required.

The basic algorithm is:

- *new bit* := bit 33 EOR bit 20
- shift left the 33 bit number
- put in *new bit* at the bottom.
- Repeat for all the 32 *new bits* needed.

All this can be done in five S cycles:

```

; Enter with seed in Ra (32 bits),Rb (1 bit in Rb lsb)
; Uses Rc
      TST     Rb,Rb,LSR #1        ; top bit into carry
      MOVS   Rc,Ra,RRX           ; 33 bit rotate right
      ADC     Rb,Rb,Rb           ; carry into lsb of Rb
      EOR    Rc,Rc,Ra,LSL#12     ; (involved!)
      EOR    Ra,Rc,Rc,LSR#20     ; (similarly involved!)
; New seed in Ra, Rb as before

```

Multiplication by a constant

Multiplication by 2^n (1,2,4,8,16,32...)

```
MOV    Ra,Ra,LSL #n;
```

Multiplication by 2^{n+1} (3,5,9,17...)

```
ADD    Ra,Ra,Ra,LSL #n.
```

Multiplication by 2^{n-1} (3,7,15...)

```
RSB    Ra,Ra,Ra,LSL #n
```

Multiplication by 6

```
ADD    Ra,Ra,Ra,LSL #1      ; Multiply by 3
MOV    Ra,Ra,LSL #1        ; and then by 2.
```

Multiply by 10 and add in extra number

```
AD     Ra,Ra,Ra,LSL #2      ; Multiply by 5
ADD    Ra,Rc,Ra,LSL #1     ; Multiply by 2 and add in next digit
```

General recursive method for $Rb := Ra \times C$, C a constant

If C even, say $C = 2^n \times D$, D odd:

```
D=1 :   MOV    Rb,Ra,LSL #n
D<>1:   {Rb := Ra*D}
        MOV    Rb,Rb,LSL #n
```

If $C \bmod 4 = 1$, say $C = 2^n \times D + 1$, D odd, $n > 1$:

```
D=1 :   ADD    Rb,Ra,Ra,LSL #n
D<>1:   {Rb := Ra*D}
        ADD    Rb,Ra,Rb,LSL #n.
```

If $C \bmod 4 = 3$, say $C = 2^n \times D - 1$, D odd, $n > 1$:

```
D=1 :   RSB    Rb,Ra,Ra,LSL #n
D<>1:   {Rb := Ra*D}
        RSB    Rb,Ra,Rb,LSL #n.
```

This is not quite optimal, but close. An example of its non-optimal use is multiply by 45 which is done by:

```
RSB    Rb,Ra,Ra,LSL #2    ; Multiply by 3
RSB    Rb,Ra,Rb,LSL #2    ; Multiply by 4*3-1 = 11
ADD    Rb,Ra,Rb,LSL #2    ; Multiply by 4*11+1 = 45
```

rather than by:

```
ADD    Rb,Ra,Ra,LSL #3    ; Multiply by 9
ADD    Rb,Rb,Rb,LSL #2    ; Multiply by 5*9 = 45
```

Loading a word from an unknown alignment

Before ARMv6, there was no instruction to load a word from an unknown alignment. To do this requires some code (which can be a macro) along the following lines:

```
; Enter with 32-bit address in Ra
; Uses Rb, Rc; result in Rd
; Note d must be less than c

BIC    Rb,Ra,#3           ; Get word-aligned address
LDMIA  Rb,{Rd,Rc}         ; Get 64 bits containing answer
AND    Rb,Ra,#3           ; Correction factor in bytes
MOVS   Rb,Rb,LSL #3       ; ..now in bits and test if aligned
MOVNE  Rd,Rd,LSR Rb       ; If not aligned, produce bottom
; of result word
RSBNE  Rb,Rb,#32          ; Get other shift amount
ORRNE  Rd,Rd,Rc,LSL Rb    ; Combine two halves to get result
```

Sign/zero extension of a half word

```
MOV    Ra,Ra,LSL #16      ; Move to top,
MOV    Ra,Ra,LSR #16      ; and back to bottom
; Use ASR to get sign extended version
```

Setting condition codes

In 26-bit modes, the PSR flags could be manipulated directly using the TEQP instruction (or its more rarely-used relations, CMNP, CMPP, and TSTP):

```
CFLAG *    &20000000
TEQP    PC,#CFLAG
```

PC, when specified in the Rn position like this, had all PSR bits clear, so this had the effect of setting C and clearing the other flags. These instructions could not discriminate between flags and other PSR bits, so the instruction also enabled all interrupts and entered USR mode (unless you were already in USR mode, which isn't privileged to make such changes).

When used in a macro, a TEQP instruction like this would often have been followed by a NOP instruction, because some CPUs had unpredictable effects if you accessed a banked register (R8-R14) in the next instruction.

PSR flag manipulation could also be combined with procedure call return in 26-bit modes like this:

```
CFLAG *      &20000000
      BICS   PC,R14,#CFLAG      ; Returns clearing C flag
                                      ; from link register
      ORRCS  PC,R14,#CFLAG      ; Conditionally returns setting C flag
```

This worked because when a procedure was entered using a BL instruction (which it usually would have been), the R14 register contained a copy of the PSR which was in use at the time. The same warnings about the effect on the interrupt-disable and processor mode bits of the PSR when using such instructions from non-USR mode apply as for TEQP.

The closest equivalent to TEQP in 32-bit modes is the MSR instruction:

```
CFLAG *      &20000000
      MSR   CPSR_f,#CFLAG
```

This can safely be used from non-USR modes without affecting CPSR bits other than the flags.

You can't combine flag manipulation with a return from a procedure call in a single instruction in 32-bit modes. You would typically replace the above example with an MSR and a MOV PC, R14.

Some CPUs that support 26-bit mode don't have the MSR instruction, and some CPUs that support 32-bit mode don't have the TEQP instruction. It is possible to construct code sequences which will detect the mode in use and use the appropriate instruction, but it is far more efficient to simply use the ALU to update the flags, since such code works on all CPUs and never affects the non-flag PSR bits. Some simple examples are given below, assuming you don't care what happens to the other flags:

To clear N:

```
TST   r0, #0
```

To set N:

```
TEQ   r0, #0
TEQPL r0, #&80000000
```

To clear Z:

```
TEQ   pc, #0
```

To set Z:

```
TEQ   r0, r0
```

To clear C:

```
CMN    r0, #0
```

To set C:

```
CMP    r0, #0
```

To clear V:

```
CMP    r0, #0
```

To set V:

```
CMP    r0, #&80000000
CMNVC  r0, #&80000000
```

Full multiply

The ARM's multiply instruction multiplies two 32 bit numbers together and produces the least significant 32 bits of the result. These 32 bits are the same regardless of whether the numbers are signed or unsigned.

To produce the full 64 bits of a product of two unsigned 32 bit numbers, the following code can be used:

```
; Enter with two unsigned numbers in Ra and Rb.
MOVVS  Rd,Ra,LSR #16      ; Rd is ms 16 bits of Ra
BIC    Ra,Ra,Rd,LSL #16  ; Ra is ls 16 bits
MOV    Re,Rb,LSR #16     ; Re is ms 16 bits of Rb
BIC    Rb,Rb,Re,LSL #16  ; Rb is ls 16 bits
MUL    Rc,RA,Rb          ; Low partial product
MUL    Rb,Rd,Rb          ; First middle partial product
MUL    Ra,Re,Ra          ; Second middle partial product
MULNE  Rd,Re,Rd          ; High partial product - NE
                          ; condition reduces time taken
                          ; if Rd is zero
ADDS   Ra,Ra,Rb          ; Add middle partial products -
                          ; could not use MLA because we
                          ; need carry
ADDCS  Rd,Rd, #&10000    ; Add carry into high partial
                          ; product
ADDS   Rc,Rc,Ra,LSL #16  ; Add middle partial product
ADC    Rd,Rd,Ra,LSR #16  ; sum into low and high words
                          ; of result

; Now Rc holds the low word of the product, Rd its high word,
; and Ra, Rb and Re hold junk.
```

Of course, newer cores provides the Multiply Long class of instructions to perform a 64 bit signed or unsigned multiply or multiply-accumulate.

Appendix E: Support for AAsm source

AAsm was an alternative variant of the assembler supplied with previous releases of this product. It has been removed from this product, but to ease porting source code written for AAsm, some limited support has been added to ObjAsm.

This support for AAsm may be removed in future releases of Acorn Assembler.

To enable this support you must pass the **--absolute** option to ObjAsm. There is no option on the Setup menu directly corresponding to this option; the best way to pass the option from the desktop is to include it in the Setup menu's **Others** option (see *Specifying other command line options* on page 26).

The --absolute option

The **--absolute** option makes ObjAsm accept AAsm source code. This option is provided to simplify the use of code originally developed using AAsm. Unlike AAsm, the output format produced is AOF, as for any ordinary assembly operation, and this must be linked by the linker as usual, in order to create an absolute image. However, the contents of the AOF file will be marked as having an absolute address (if either the **ORG** or **LEADR** directive is used), and the linker, given suitable options, can produce an image file equivalent to that previously generated directly by AAsm. The following changes to normal ObjAsm input syntax apply:

- There is an implicit **AREA** declaration before the start of the source. The normal rule that there must be an **AREA** directive in the source before use of any instruction or data generating statements does not apply. The implicitly declared area is called **ABS\$\$BLOCK**, and has the **ABS** attribute (see *Area attributes* on page 42) implying that it must be loaded at a fixed absolute base address.
- The directive **LEADR** is accepted. (Previously only AAsm implemented this; ObjAsm did not.)
- The **ORG** directive, if used within the source file, will apply to the implicitly declared current area.
- The following directives are not recognised (since they were not available with AAsm), and may be used for any other purpose, in particular as macro names: **AOF**, **AOUT**, **AREA**, **ENTRY**, **EXPORT**, **EXPORTAS**, **EXTERN**, **GLOBAL**, **IMPORT**, **KEEP**, **REQUIRE**, **STRONG**.

This change is important, since ObjAsm recognises directives before it does macro names.

Index

Symbols

! (directive) 70
! (operator) 81
!= (operator) 85
(directive) 66
#include 12
\$ (macro parameter reference) 94
\$ (variable substitution) 75
% (directive) 63
% (local label reference) 46
% (operator) 83
& (directive) 63
& (operator) 84
&& (operator) 85
* (directive) 73
* (operator) 83
+ (binary operator) 84
+ (unary operator) 81
. (built-in variable) 77
/ (operator) 83
/= (operator) 85
< (operator) 85
<< (operator) 84
<= (operator) 85
<> (operator) 85
= (directive) 63
= (operator) 85
== (operator) 85
> (operator) 85
>< (operator) 85
>= (operator) 85
>> (operator) 84
? (operator) 81
@ (built-in variable) 66, 80
| (directive) 87-89
| (directive) 87-89

^ (directive) 66
^ (operator) 84
__RelocCode 104
| (directive) 87-89
| (operator) 84
|| (operator) 85
lobjasm\$version| 80

A

AAsm 36, 145
ABS 42, 43, 145
accn *see* registers (names)
ADC 49
ADD 49, 50, 51
ADR 50
ADRL 51
Advanced SIMD 56, 74, 78
ALIAS 75
ALIGN 43, 72
an *see* registers (names)
AND 49, 84
AOF 44, 145
AOUT 44, 145
APCS 15, 17, 33, 73, 107, 108
 qualifiers
 /26bit 34, 76
 /32bit 34, 76
 /fp 33, 34
 /fpa 35, 36
 /fpe2 35
 /fpe3 35
 /fpregargs 35
 /hardfp 35, 36
 /interwork 35, 43, 76
 /nofp 34

- /nofpregargs 36
- /nointerwork 35
- /nonreentrant 33
- /nopic 33
- /nopid 33
- /noropi 33
- /norwpi 33, 69
- /noswstackcheck 34, 43
- /pic 33
- /pid 33
- /reentrant 33, 42, 77
- /ropi 33, 77
- /rwpi 33, 69, 77
- /softfp 35
- /swstackcheck 33, 34
- /vfp 35, 43, 60, 65

ARCHITECTURE 76

AREA 42, 104, 145

AREANAME 76

AREAs 33, 41-43, 72

- !\$\$\$\$\$\$! 42
- !ABS\$\$BLOCK! 145
- !C\$\$codel 42
- attributes 42
- code 41
- data 41
- relocatable address constants 42

ARM

- configuration 34
- directive 44
- EQU attribute 73
- EXPORT qualifier 68
- versions 2, 16

ARM ARM 3

ARM Procedure Call Standard *see* APCS

AsmHello example 29

AsmModule example 105

assembly language 39-97

- examples 139-144

ASSERT 70

B

BASE 82

BASED 69

BASED R_n 43

bibliography 3

BIC 49

booleans *see* constants

built-in variables 76

buttons *see* application (*button name*)

C

C language 107-110

- preprocessor 12, 13
- static variables 109-110
- strings 64

cacheing *see* ObjAsm (cacheing)

case sensitivity 15, 41, 45

CC 83

CC_ENCODING 82

changes 113

CHR 82

CLZ 77

CMN 49

CMP 49

CN 74

cn *see* registers (names)

CODE 42, 68

CODE16

- directive 44
- EQU attribute 73

CODE32

- directive 44
- EQU attribute 73

CODEALIGN 43, 72

CODESIZE 76

COMDEF 42

comments 47

COMMON 42

condition codes 139-140, 142-144

conditional assembly 22, 87-89

CONFIG 34, 76
 constants 47, 73
 immediate 49
 VFP 58
 conventions 4
 coprocessors 55, 74
 Cortex-A8 101
 CP 74
 CPU 76
 CStatics example 109-110

D

DATA
 AREA attribute 42
 directive 45
 EQU attribute 73
 EXPORT/IMPORT qualifier 68
 DCB 63
 DCD 63
 DCDO 63, 64
 DCDU 63
 DCFD 65
 DCFDU 65
 DCFH 65
 DCFHU 65
 DCFS 65
 DCFSU 65
 DCI 63, 64
 DCQ 63
 DCQU 63
 DCW 63
 DCWU 63
 DDT 12
 debugging 12
 machine level 12
 source level 12
 tables 12
 DEF 82
 dependency lists 36
 dialogue boxes *see application (dialogue box name)*
 directives 41, 45, 63-72, 145

see also directive name

DN 74
Dn and dn see registers (names)

E

ELIF 87-89
 ELSE 87-89
 END 47, 66
 ENDIAN 32, 76
 ENDF 87-89
 ENTRY 72, 104, 145
 EOR 84
 EQU 73
 errors 12, 20, 28, 70, 123-135
 browser 12, 28
 fatal 123
 escapes 15
 EXPORT 67, 145
 EXPORTAS 69, 145
 expressions 81-85
 external 63
 EXTERN 67, 145

F

FALSE 47, 76
 FCONST 60
 FIELD 66
 FILL 63, 64
 FLD 60
 floating point 55-62, 72, 103
 calling standard 35-36
 emulator 55
 number input 56
 FN 73
Fn and fn see registers (names)
fp see registers (names)
 FP3 35
 FPA 17, 35, 55, 73, 78, 79
 FPREGARGS 68

FPU 76
frame pointer 34

G

GBL 36, 44, 74
GET 12, 25, 66, 69
GLOBAL 68, 145

H

HALFWORD 43
HARDFP 68

I

icons *see application (icon name)*
IF 87-89
image files 9, 12, 29
IMPORT 67, 104, 145
INCLUDE 12, 69
include file searching 12
INDEX 82
INFO directive 70
initialising memory *see memory (initialising)*
INPUTFILE 76
installation 1
instructions
 conversions 49
 single data transfer 43
 software interrupt 103
 undefined 55
INTER 35, 76
interrupt handlers 102
INTERWORK 43
ip *see registers (names)*
IT 51

K

KEEP 66, 145

L

labels 41, 45
 local 46
LAND 85
layout of memory *see memory (laying out)*
LCL 44, 74, 93
LDC 60
LDF 60
LDM 72
LDR 52
LDRB 52
LDRD 52, 77
LDRH 52
LDRSB 52
LDRSH 52
LEADR 145
LEAF 66, 68
LEFT 83
LEN 82
LEOR 85
libraries 9
LINENUM 76
LINENUMUP 76
LINENUMUPPER 76
Link 2, 9, 34, 41
 Debug 12
 Module 105
listings 22-24, 89
 options 71
literals 50, 52, 60, 67
LNK 25
LNOT 82
LOR 85
LOWERCASE 82
LR 103
lr *see registers (names)*
LTORG 60, 66

M

MACRO 92-93
macros 89, 91-97, 145
 for manipulating PSR 101
 labels 46
 names 15
 nesting 95
 parameters 92, 94-95
 prototype statements 92-93
Make 9, 30, 36
MAP 66
MaverickCrunch 55
memory
 initialising 63-65
 laying out 66
 reserving 63
MEND 71, 93
menus *see application (menu name)*
MEXIT 94
MNF 60
MOD 83
modules 9, 103-105
MOV 49, 50, 51, 52
MOV32 50, 51
MOVH 50
MOVW 49, 50, 52
MRS 101
MSR 101
multiplication 141-142, 144
 see also instructions (multiplies)
MVF 60
MVN 49, 50, 51, 52

N

NEON 56
NOFP 72
NOFPREGARGS 68
NOINIT 42
NONLEAF 68
NOSWSTACKCHECK 43

NOT 82
NOUSESSB 68
NOWEAK 68
numbers *see constants*

O

ObjAsm 2, 9-37
 All registers 15
 APCS Registers 15
 ARM only 20
 Auto run 28
 Auto save 28
 C strings 15
 Cache size 20
 Cache source 20
 cacheing 19
 Check register lists 20
 command line 26, 30-37
 Command line (menu option) 14
 CPU 16
 Cross reference 24
 Debug 12
 Device 18
 Display 28
 Errors to file 21
 FPU 17
 Help 29
 icon bar menu 28
 Include 12
 Keep symbols 21
 Length 24
 Listing 22
 No APCS registers 15
 No code generation 12
 No registers 15
 Options 28
 Others 26
 output 27-28
 Pre-UAL ARM 13
 Pre-UAL Thumb 13
 Run 11, 14

- Save options 28
- SetUp dialog box 9, 10-12
- SetUp menu 11
- Source 10, 11
- Suppress warnings 20
- Terse listing 22, 89
- Throwback 12
- UAL ARM 13
- Upper case opcodes 15
- Use C preprocessor 13
- Width 23
- Work directory 25
- object files 9, 29, 41, 63, 67
- operators 81-85
 - addition and logical 84
 - binary 83-85
 - boolean 85
 - multiplicative 83
 - precedence 81, 83
 - relational 85
 - shifts 84
 - string manipulation 83
 - unary 81-83
- OPT 71, 76
- OR 84
- ORG 43, 51, 66, 145
- origin 67
- output 27, 28

P

- PC 72, 77
- pc *see* registers (names)
- PCSTOREOFFSET 77
- PIC 42
- pn *see* registers (names)
- POP 72
- PrintLib example 107-109
- PSR manipulation 101
- PUSH 72

Q

- QADD 78
- QN 74
- Qn and qn *see* registers (names)

R

- random numbers 140
- RCONST 82
- READONLY 42, 68
- READWRITE 42, 69
- REENTRANT 33, 42, 77
- registers
 - names 15, 33, 41, 73
- REL 42
- relocatable modules *see* modules
- relocations 63
- repetitive assembly 90
- REQUIRE 69, 145
- reserving memory *see* memory (reserving)
- REVERSE_CC 82
- RIGHT 83
- RISC OS 99-110
- RLIST 72
- RN 73
- Rn and rn *see* registers (names)
- ROL 84
- ROPI 33, 77
- ROR 84
- ROUT 45, 46
- RWPI 33, 77

S

- sb *see* registers (names)
- SBC 49
- SDIV 77
- SET 36, 44, 74, 94
- SHL 84
- SHR 84

- sign extension 142
- sl *see* registers (names)
- SMULL 78
- SN 73
- Sn and sn see* registers (names)
- SOFTFP 68
- SoftFP 35, 79
- SoftFPA 17
- SoftVFP 17
- source files 69
 - line length 41
- sp *see* registers (names)
- SPACE 63, 64
- SrcEdit 28
- stack-limit checking 34
- STM 72
- STR 75, 83
- strings *see* constants
- STRONG 69, 145
- Strong ARM 101
 - bug 102
- SUB 49, 50, 51
- SUBT 71
- summary 27, 28
- SVC mode 103
- SWI 103
- symbols 24, 44, 53, 67, 73-76
 - length 45
 - local 67
- TARGET_FEATURE_NEON 78
- TARGET_FEATURE_NEON_FP16 78
- TARGET_FEATURE_NEON_FP32 78
- TARGET_FEATURE_NEON_INTEGER 78
- TARGET_FEATURE_UNALIGNED 78
- TARGET_FPU_FPA 78
- TARGET_FPU_SOFTFPA 79
- TARGET_FPU_SOFTFPA_FPA 79
- TARGET_FPU_SOFTFPA_VFP 79
- TARGET_FPU_SOFTVFP_FPA 79
- TARGET_FPU_SOFTVFP_VFP 79
- TARGET_FPU_VFP 79
- TARGET_FPU_VFPV2 79
- TARGET_FPU_VFPV3 79
- TARGET_FPU_VFPV4 79
- TARGET_PROFILE_A 79
- TARGET_PROFILE_M 80
- TARGET_PROFILE_R 80
- throwback 28
- THUMB
 - EQU attribute 73
 - EXPORT qualifier 68
- Thumb
 - ARM/Thumb interworking 35, 43, 44
- titles 71
- tools 7-37
 - common features 9, 27
- TRUE 47, 80
- TTL 71
- typographic conventions *see* conventions

T

- TARGET_ARCH_ARM 77
- TARGET_ARCH_THUMB 77
- TARGET_FEATURE_CLZ 77
- TARGET_FEATURE_DIVIDE 77
- TARGET_FEATURE_DOUBLEWORD 77
- TARGET_FEATURE_DSPMUL 78
- TARGET_FEATURE_EXTENSION_REGISTER_COUNT 78
- TARGET_FEATURE_MULTIPLY 78
- TARGET_FEATURE_MULTIPROCESSING 78

U

- UAL 3, 13, 44, 80
- UDIV 77
- UND 52
- UPPERCASE 83
- USESSB 68

V

VAND 58
VAR 80
variables 36, 73-76
 global 74
 local 74, 93
VBIC 58, 59
VFP 17, 35, 43, 55, 73, 78, 79
VLDR 60
VMOV 58, 59, 60
VMVN 58, 59
vn *see* registers (names)
VORN 58
VORR 58, 59
VTRN 59
VUZP 59
VZIP 59

W

warnings 20, 135-138
WEAK 68
WEND 90, 94
WHILE 90, 94
Wireless MMX 55
work directory 25

X

XScale 53, 101