ARM® Developer Suite

Version 1.1

Getting Started

ARM

Copyright © 1999, 2000 ARM Limited. All rights reserved.
ARM DUI 0064C

Getting Started
Version 1.1

Copyright © 1999 and 2000 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
October 1999	A	Release 1.0
March 2000	B	Release 1.0.1
October 2000	C	Release 1.1

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Contents
Getting Started

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Preface

This preface introduces the *ARM Developer Suite* (ADS) and its user documentation. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page ix.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

About this book

This book provides an overview of the ADS tools and documentation.

Intended audience

This book is written for all developers who are producing applications using ADS. It assumes that you are an experienced software developer.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to ADS. The components of ADS and the printed and online documentation are described.

Chapter 2 ***Differences***

Read this chapter for details of the differences between versions of ADS and the ARM Software Development Toolkit.

Chapter 3 ***Creating an application***

Read this chapter for a brief overview of how to create an application using the command-line tools or the CodeWarrior IDE.

Chapter 4 ***Migrating Projects from SDT to ADS***

Read this chapter for information on converting an SDT 2.50/2.51 project to ADS.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Typographical conventions

The following typographical conventions are used in this book:

`typewriter` Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

`typewriter italic`

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists, where appropriate, and for ARM processor signal names.

`typewriter bold`

Denotes language keywords when used outside example code.

Further reading

This section lists publications from ARM Limited that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at `http://www.arm.com/DevSupp/Sales+Support/faq.html`

ARM publications

Refer to the following books in the ADS document suite for information on other components of ADS 1.1:

- *ADS Installation and License Management Guide* (ARM DUI 0139)
- *ADS Assembler Guide* (ARM DUI 0068)
- *CodeWarrior IDE Guide* (ARM DUI 0065)
- *ADS Compiler, Linker, and Utilities Guide* (ARM DUI 0067)
- *ADS Debuggers Guide* (ARM DUI 0066)

Copyright © 1999, 2000 ARM Limited. All rights reserved.

- *ADS Debug Target Guide* (ARM DUI 0058)
- *ADS Developer Guide* (ARM DUI 0056).

The following additional documentation is provided with the ARM Developer Suite:

- *ARM Architecture Reference Manual* (ARM DDI 0100). This is supplied in DynaText and PDF format.
- *ARM Applications Library Programmer's Guide*. This is supplied in DynaText and PDF format.
- *ARM ELF specification* (SWS ESPC 0003). This is supplied in PDF format in `install_directory\PDF\specs\ARMELF.pdf`.
- *TIS DWARF 2 specification*. This is supplied in PDF format in `install_directory\PDF\specs\TIS-DWARF2.pdf`.
- *ARM/Thumb® Procedure Call Standard specification*. This is supplied in PDF format in `install_directory\PDF\specs\ATPCS.pdf`.

In addition, refer to the following documentation for specific information relating to ARM products:

- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Feedback

ARM Limited welcomes feedback on both the ARM Developer Suite and its documentation.

Feedback on the ARM Developer Suite

If you have any problems with the ARM Developer Suite, please contact your supplier. To help them provide a rapid and useful response, please give:

- details of the release you are using
- details of the platform you are running on, such as the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tool, including the version number and date.

Feedback on this book

If you have any problems with this book, please send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem.

General suggestions for additions and improvements are also welcome.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Chapter 1
Introduction

This chapter introduces the *ARM Developer Suite version 1.1* (ADS 1.1) and describes its software components and documentation. It contains the following sections:

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

1.1 About the ARM Developer Suite

ADS consists of a suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors.

You can use ADS to develop, build, and debug C, C++, or ARM assembly language programs.

1.1.1 Components of ADS

ADS consists of the following major components:

- *Command-line development tools*
- *GUI development tools* on page 1-3
- *Utilities* on page 1-3
- *Supporting software* on page 1-4.

Command-line development tools

The following command-line development tools are provided:

armcc The ARM C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI source into 32-bit ARM code.

armcpp This is the ARM C++ compiler. It compiles ISO C++ or EC++ source into 32-bit ARM code.

tcc The Thumb C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI source into 16-bit Thumb code.

tcpp This is the Thumb C++ compiler. It compiles ISO C++ or EC++ source into 16-bit Thumb code.

armasm The ARM and Thumb assembler. This assembles both ARM assembly language and Thumb assembly language source.

armlink The ARM linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program. The ARM linker creates ELF executable images.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

armsd The ARM and Thumb symbolic debugger. This enables source level debugging of programs. You can single-step through C or assembly language source, set breakpoints and watchpoints, and examine program variables or memory.

Rogue Wave C++ library

The Rogue Wave library provides an implementation of the standard C++ library as defined in the *ISO/IEC 14822:1998 International Standard for C++*. For more information on the Rogue Wave library, see the online HTML documentation on the CD ROM.

support libraries

The ARM C libraries provide additional components to enable support for C++ and to compile code for different architectures and processors.

GUI development tools

The following *Graphical User Interface* (GUI) development tools are provided:

AXD The ARM Debugger for Windows and UNIX. This provides a full Windows and UNIX environment for debugging your C, C++, and assembly language source.

ADW The ARM Debugger for Windows. This provides a full Windows environment for debugging your C, C++, and assembly language source.

ADU The ARM Debugger for UNIX. This provides a full GUI environment for debugging your C, C++, and assembly language source.

CodeWarrior IDE

The project management tool for Windows. This automates the routine operations of managing source files and building your software development projects. The CodeWarrior IDE is not available for UNIX.

Utilities

The following utility tools are provided to support the main development tools:

fromELF The ARM image conversion utility. This accepts ELF format input files and converts them to a variety of output formats, including plain binary, *Extended Intellec Hex* (IHF) format, Motorola 32-bit S-record format, Intel Hex 32 format, and Verilog Hex format. fromELF can also generate text information about the input image, such as code and data size.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

armprof The ARM profiler displays an execution profile of a simple program from a profile data file generated by an ARM debugger.

armar The ARM librarian enables sets of ELF format object files to be collected together and maintained in libraries. You can pass such a library to the linker in place of several ELF files.

Flash downloader

Utility for downloading binary images to Flash memory on an ARM Development board (PID7T) or an ARM Integrator™ board.

Supporting software

The following support software is provided to enable you to debug your programs, either under simulation, or on ARM-based hardware:

ARMulator® The ARM core simulator. This provides instruction-accurate simulation of ARM processors, and enables ARM and Thumb executable programs to be run on non-native hardware. The ARMulator is integrated with the ARM debuggers.

Supported standards

The industry standards supported by ADS include:

ar UNIX-style archive files are supported by armar.

DWARF2 DWARF2 debug tables are supported by the compilers, linker, and debuggers. The deprecated format DWARF1 is supported in the debuggers only.

ANSI C The ARM and Thumb compilers accept ANSI C as input. The option `-strict` can be used to enforce strict acceptance.

C++ The ARM and Thumb C++ compilers support a subset of the ISO C++ language.

EC++ The ARM and Thumb C++ compilers support the *Embedded C++* (EC++) informal standard that is a subset of C++.

ELF The ARM tools produce ELF format files. The FromELF utility can translate ELF files into other formats.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

RDI All debug agents and targets within ADS support version 1.5.1 of the *Remote Debug Interface* (RDI). The debuggers support all the debug agents (for example ARMulator and Remote_A) that are released as part of ADS. They also support Multi-ICE®.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

1.2 Printed documentation

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the ARM Frequently Asked Questions list.

1.2.1 ADS publications

This book contains general information about ADS. Other publications included in the suite are:

- *ADS Installation and License Management Guide* (ARM DUI 0139). This book gives detailed installation and license management information. It describes how to install ADS, how to install license files for ADS, and how to work with the FLEX*lm* license management system.
- *ADS Assembler Guide* (ARM DUI 0068). This book provides reference and tutorial information on the ARM assembler.
- *ADS Compiler, Linker, and Utilities Guide* (ARM DUI 0067). This book provides reference information for ADS. It describes the command-line options to the assembler, linker, compilers, and other ARM tools in ADS. The book also gives reference material on the ARM implementation of the C and C++ compilers and the C libraries.
- *ADS Developer Guide* (ARM DUI 0056). This book provides tutorial information on writing code targeted at the ARM family of processors
- *ADS Debuggers Guide* (ARM DUI 0066). This book has three main parts in which all the currently supported ARM debuggers are described:
  - Part A describes the graphical user interface components of *ARM eXtended Debugger* (AXD), the most recent ARM debugger and part of the ARM Developer Suite of software. Tutorial information is included to demonstrate the main features of AXD.
  - Part B describes the *ARM Debugger for Windows* (ADW) and the *ARM Debugger for UNIX* (ADU). These are provided for backwards compatibility and will not be supported in future releases.
  - Part C describes the *ARM Symbolic Debugger* (armsd).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

- *ADS Debug Target Guide* (ARM DUI 0058). This book provides reference and tutorial information on the debug targets that can be used with the ARM debuggers. In particular, it describes the ARMulator, the ARM instruction set simulator, in detail.
- *CodeWarrior IDE Guide* (ARM DUI 0065). This book provides tutorial and reference information on the CodeWarrior Integrated Development Environment. The CodeWarrior IDE is used to manage C, C++, and assembly language projects in ADS. The CodeWarrior IDE and guide are available only on Windows.

See also the Further Reading sections in each book for related publications from ARM, and from third parties.

Additional documentation is supplied as PDF files in the PDF subdirectory of your ADS installation directory.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

1.3 Online documentation

The ADS printed documentation is also available online as DynaText electronic books. The content of the DynaText manuals is identical to that of the printed and PDF documentation.

In addition, documentation for the Rogue Wave C++ library is available in HTML format. See *HTML* on page 1-14 for more information.

PDFs of the ADS manuals are installed only for a Full installation. The Typical installation only installs PDFs of related documentation that is not available in the printed books or DynaText online books.

1.3.1 DynaText

The manuals for ADS are provided on the CD-ROM as DynaText electronic books. The DynaText browser is installed by default for a Typical or Full installation.

To display the online documentation, either:

- select **Online Books** from the **ARM Developer Suite v1.1** program group
- execute `install_directory\dtext41\bin\Dtext.exe`.

The DynaText browser displays a list of available collections and books (Figure 1-1).

Figure 1-1 DynaText browser with list of available books

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Opening a book

Double-click on a title in the book list to open the book. The table of contents for the book is displayed in the left panel and the text is displayed in the right panel (see Figure 1-2).

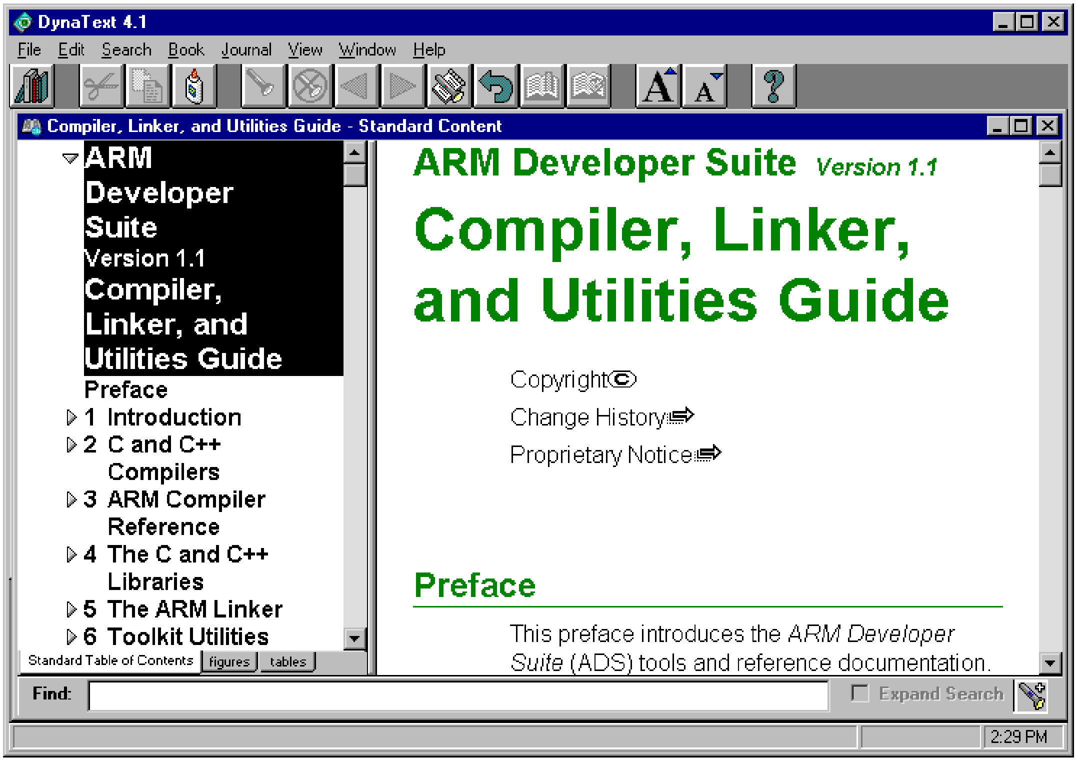

Figure 1-2 Opening a book

Navigating through the book

Click on a section in the table of contents to display the text for that section. For example, selecting *C and C++ libraries* displays the text for that section (see Figure 1-3).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

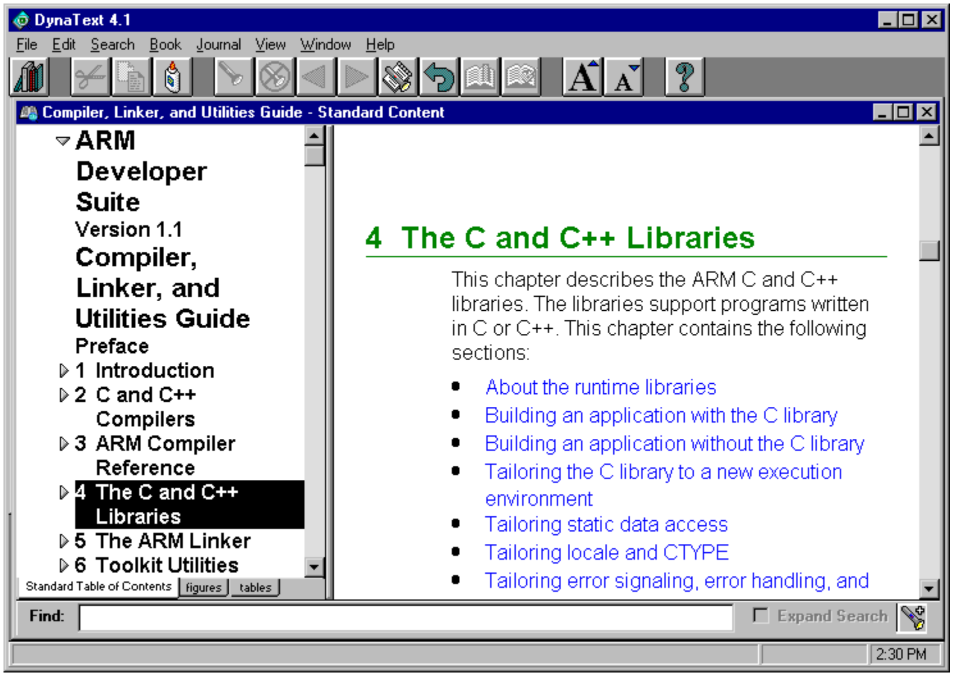

Figure 1-3 Selecting a section from the table of contents

Navigating using hyperlinks

Text in blue indicates a link that displays a different section of a book, or a different book. Plain blue text indicates that the link is within the current chapter. Underlined blue text indicates that the link is either to another chapter within the current book, or to a different book. Hyperlinks behave differently depending on their target:

- if the link is within the current chapter (plain blue text), DynaText scrolls the current window to display the target
- if the link is to another chapter in the current book, DynaText opens a new window without a Table of Contents
- if the link is to another book, DynaText opens a new window with a Table of Contents.

Figure 1-4 shows the browser displaying the text for the linked text.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

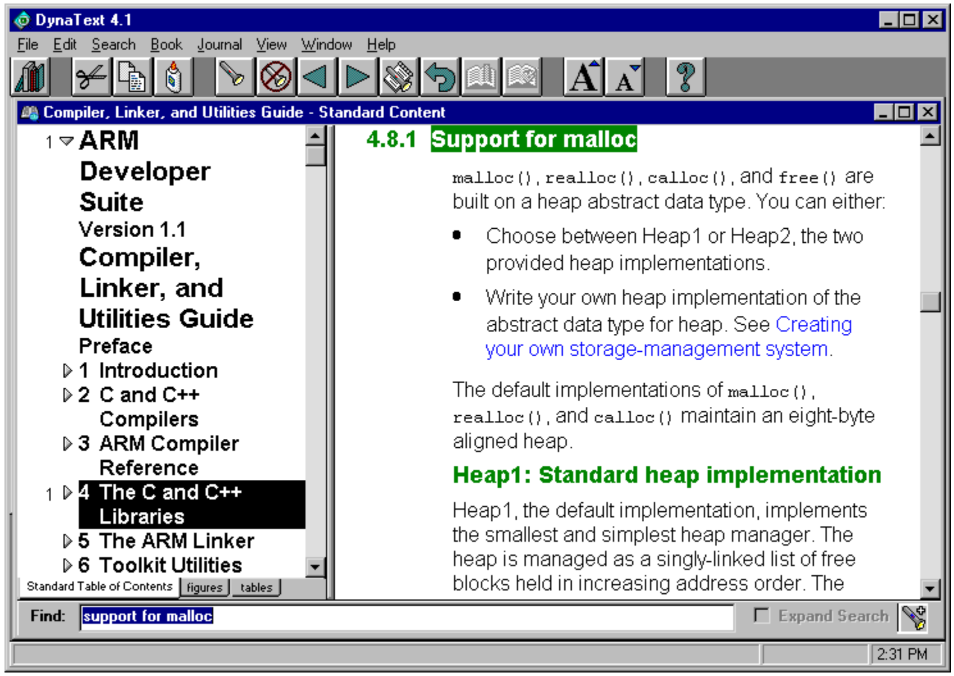

Figure 1-4 Using text links

Displaying graphics

Graphics are not displayed inline in the DynaText browser. If a graphic symbol is displayed, select it to display the linked graphic in its own window (see Figure 1-5).

Figure 1-5 Link to a figure

Clicking on the figure icon displays the figure in its own window (see Figure 1-6).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

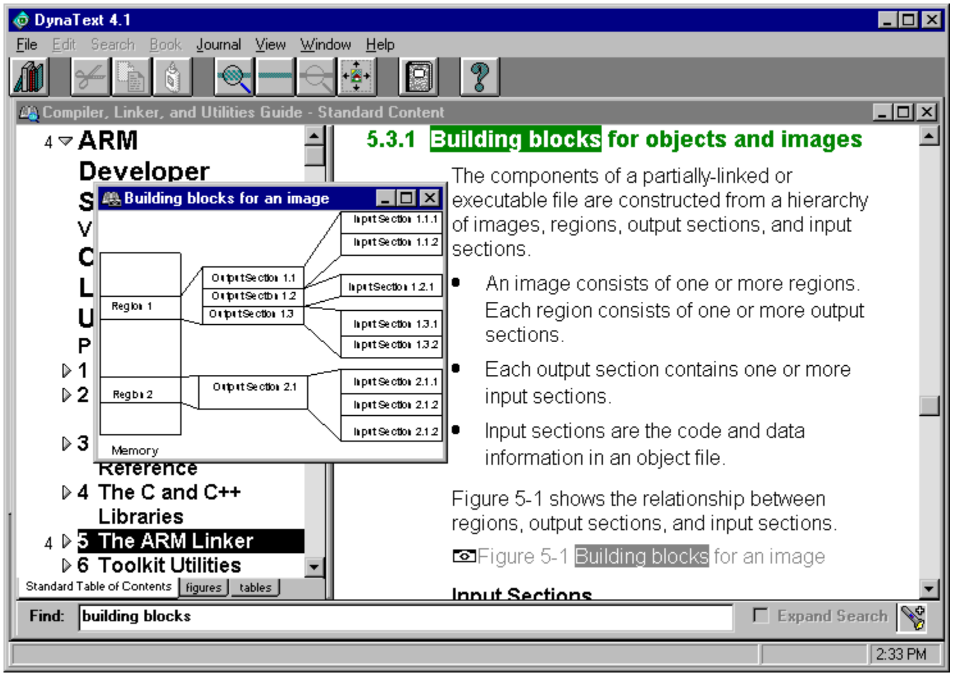

Figure 1-6 Graphic displayed

Navigating to a different book

If the blue link text refers to a different book, clicking on the link text displays the linked book in its own window (see Figure 1-7 on page 1-13).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

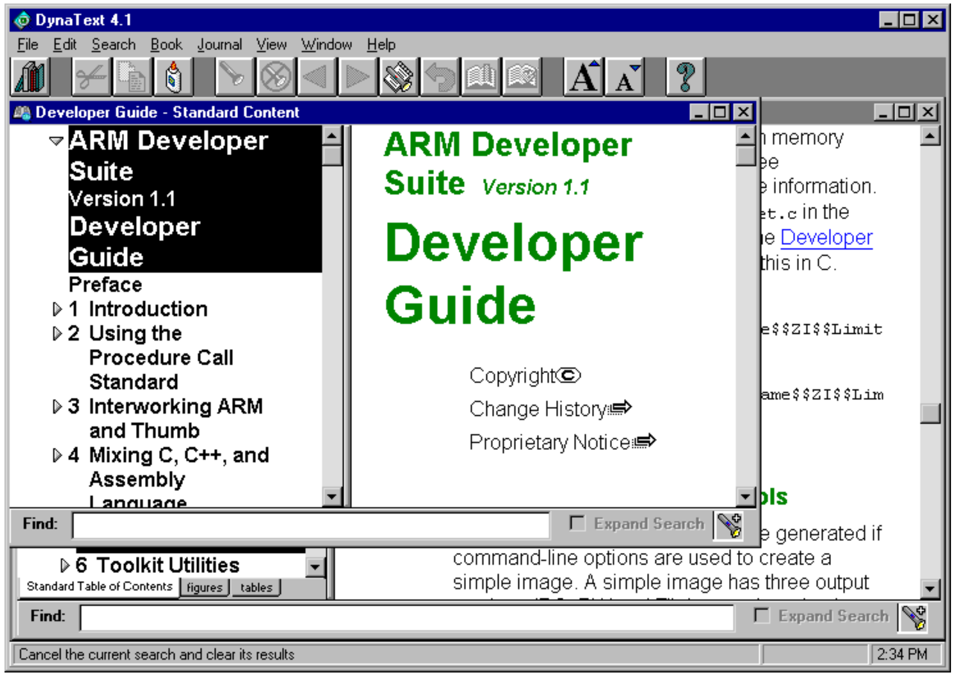

Figure 1-7 Navigating to a different book

Displaying help for DynaText

Select **Help** → **Reader Guide** to display help on how to use DynaText.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

1.3.2 HTML

The manuals for the Rogue Wave C++ library for ADS are provided on the CD-ROM in HTML files. Use a web browser, such as Netscape Communicator or Internet Explorer, to view these files. For example, select `install_directory\Html\stdref\index.htm` to display the HTML documentation for Rogue Wave (see Figure 1-8).

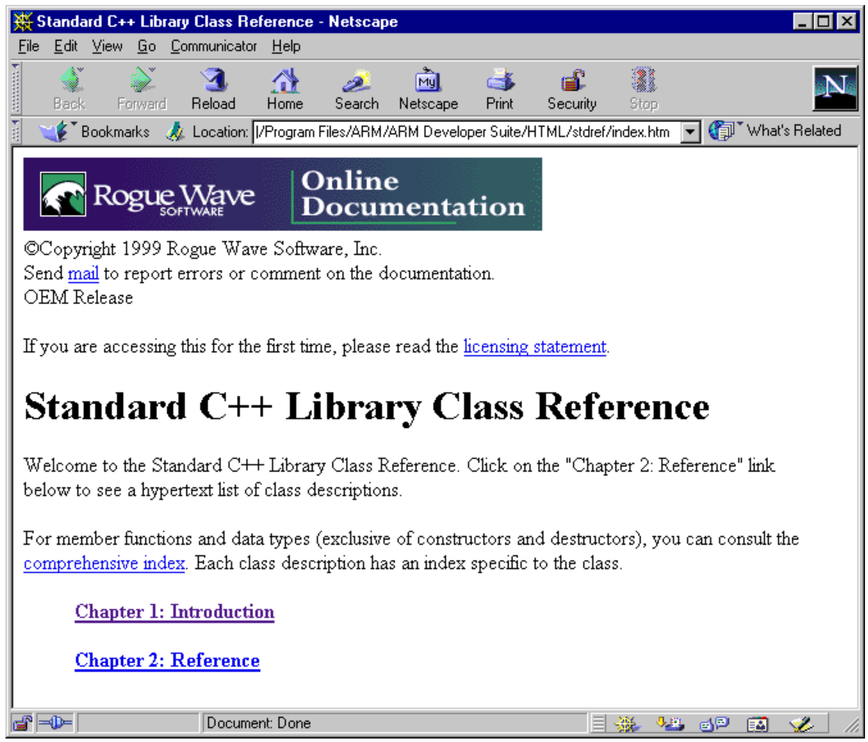

Figure 1-8 HTML browser

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

1.4 Online help

Additional information for ADS is available online. The online documentation consists of online help, DynaText files, PDF files, and HTML files.

The printed documentation for ADS is also available in DynaText and PDF files. There is also additional documentation available online that is not part of the printed documentation.

1.4.1 Online help

A **Help** menu is available for the Graphical User Interface components of ADS.

Select **Help** → **Contents** to see a display of the main help topics available. The CodeWarrior IDE does not have a **Contents** menu item. Use the **How to** menu item instead.

You can navigate to a particular page of help in any one of the following ways:

- From the **Contents** tab of the Help Topics screen, do any of the following:
  - click on a main topic to select it
  - click on the **Open** button
  - click on a subtopic.
- From the **Contents** tab of the Help Topics screen either:
  - double-click on a main topic book to open it (single-clicking toggles the open or closed status)
  - click on a subtopic.
- From the **Index** tab of the Help Topics screen, do any of the following:
  - type the first few characters of a likely index entry
  - scroll down the displayed list of index entries until the entry you want is visible
  - click on the required index entry.
- From the **Find** tab of the Help Topics screen, do any of the following:
  - type or select key words that might occur anywhere in the help text
  - select a topic from the displayed list of topics that contain the specified words.
- From any page of help that has a hypertext link to the page you want, click on the highlighted hypertext link.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

- Most pages of online help contain help links that can be clicked on:
  - — highlighted hot spots with dashed underlining display brief explanations in pop-up boxes
  - — highlighted hot spots with solid underlining jump to other related pages of help
  - — browse buttons display related pages of help.

Note

Most help selections can be done by key presses or mouse clicks.

Context-sensitive help

Context-sensitive help is available where appropriate. With the ADS component running, position the cursor on any field or button for which you need help and press the F1 key on the keyboard. If relevant online help is available it is displayed.

An alternative method of invoking context-sensitive help is to click on the question mark tool in the toolbar, then click on the field or button for which you need help.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Chapter 2
Differences

This chapter describes the major differences between the SDT 2.50/2.51, ADS 1.0, and ADS 1.1. It contains the following sections:

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.1 Overview

This chapter describes the changes that have been made between SDT 2.50/2.51 and ADS 1.0, and between ADS 1.0 and ADS 1.1.

Note

Changes made for the ADS 1.0.1 maintenance release are described in the ADS 1.0.1 Addenda. The functional changes made for ADS 1.0.1 are minor. The most important are:

- improvements to the AXD user interface
- the default stack checking option for the assembler was changed to `/noswst`, to match the compilers.

Changes made to the ARMulator for ADS 1.0.1 are superseded by changes for ADS 1.1.

The most important differences between ADS 1.1 and ADS 1.0 are:

- The ATPCS now requires 8-byte alignment of the stack.
- Full support for ARM9E™ and ARM10™
- Support for ARM architecture v5TE processors, including the Intel XScale.
- Improved code size for compiled code.
- Improved debug view for compiled code.
- Support for RealMonitor.
- Angel™ has been moved to the ARM Firmware Suite.
- The ARMulator rebuild kit is no longer supplied. It is replaced with a new ARMulator extension kit. The ARMulator has been integrated into a single DLL (shared object under UNIX) and its configuration has been simplified. A new interface enables addition of memory models.
- Limited support for GNU images in AXD.
- More components are license-managed, including the CodeWarrior IDE, fromELF, and armsd.

The most important differences between ADS 1.0 and SDT 2.50/2.51 are:

- C and C++ libraries are supplied as binaries only. Selection of the appropriate library for the build option is automatic. No rebuild kit or source code is supplied.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

The C libraries are suitable for embedded applications.

- The CodeWarrior IDE is used for project management instead of APM.
- AXD is a new debugger for Windows and UNIX. ADW for Windows and ADU for UNIX are still supported.
- AXD supports the new RDI 1.51 release.
- armar replaces armlib as library manager and ar format replaces ALF as the library format.
- The image format is now ELF.
- The preferred and default debug table format is DWARF2.
- Support for ARM9E™ and preliminary support for ARM10™.
- Major components are licence managed.
- Manuals are provided in DynaText form for easy browsing.
- A new *ARM/Thumb Procedure Call Standard* (ATPCS) encompasses ARM and Thumb on an equal basis.
- The included C++ compilers are fully integrated with ADS, and include support for Embedded C++.
- ARMulator supports RPS Interrupt Controller and Timer peripheral models.
- Clearer messages have been provided in many of the tools.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.2 Changes between ADS 1.1 and ADS 1.0

This section describes changes between ADS 1.1 and ADS 1.0. It contains the following subsections:

- *Functionality enhancements and new functionality* on page 2-4
- *Differences in default behavior* on page 2-10
- *Changes to the compilers and libraries* on page 2-10
- *Changes to the assembler* on page 2-14
- *Changes to the linker* on page 2-16
- *Changes to fromELF* on page 2-17
- *Changes to the debuggers* on page 2-18
- *Changes to ARMulator* on page 2-20.
- *Changes to the CodeWarrior IDE* on page 2-22.

2.2.1 Functionality enhancements and new functionality

This section gives a summary of the major functionality enhancements in ADS 1.1. See the sections for the individual tools for a more detailed description of changes and enhancements for each component of ADS.

ADS 1.1 introduces the following functionality enhancements and new functionality:

- *Support for ARM architecture v5TE* on page 2-4
- *Improved debug view* on page 2-4
- *Code size improvements and improved optimization* on page 2-7
- *Support for RealMonitor* on page 2-7
- *Changes to memory alignment* on page 2-8
- *Angel moved to AFS* on page 2-9.

Support for ARM architecture v5TE

ADS 1.1 fully supports ARM architecture v5TE.

Improved debug view

The reliability of the debug view in the ADS debuggers has been substantially improved, especially for optimization level `-O1`. Improvements to DWARF2 support enable you to:

- Debug inline functions.
- View return values for functions.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

- Reliably examine the contents of variables. Where the value of a variable is unavailable, it is described as such in the debugger.
- Reliably set watchpoints on local variables.
- Set breakpoints on closing parentheses to functions.
- Set breakpoints on multiple statements on the same source line.

Note

The debug view is dependent on the optimization level selected. In addition, there are some restrictions to the debug view for ADW/ADU and armsd.

Improved support for debugging third party images

AXD can now load and, with limitations, debug ELF/DWARF images built with the GNU toolchain. The following restrictions apply to using AXD with gcc 2.95.2 and binutils 2.10:

- Binutils does not set the ELF flag to indicate that an entry point has been set. You must manually set the PC to the entry point for the image. This is commonly `0x00008000` or `0x0`.
- Binutils does not generate the ARM mapping symbols that distinguish between ARM code (`$a`), Thumb code (`$t`), and data (`$d`). This means that:
  - You must manually select the disassembly mode in the disassembly window.
  - Interleaved source and code is not disassembled. It is treated as word-sized data.
  - You cannot single step, because AXD cannot determine whether to set an ARM breakpoint or a Thumb breakpoint.

    Note

    You can manually set an ARM breakpoint, however the debugger requests that you confirm the action because it interprets the code as being a literal pool.

  You can manually add a mapping symbol to mark ARM or Thumb state code by linking the following assembly language at the start of your image. If you are using the ARM assembler:

  ```
  CODE32      ; or CODE16 for Thumb
  AREA ||.text||, CODE, READONLY
  NOP
  END
  ```

  If you are using the GNU assembler:

Copyright © 1999, 2000 ARM Limited. All rights reserved.

```
    .text
        .type    $a,function    @ or $t for Thumb
$a:
        nop
```

The mapping symbol is in effect for the rest of the image, or until another mapping symbol is encountered.

This provides a workaround for the disassembly and stepping restrictions listed above for images that contain only ARM code or only Thumb code. However, it means that literal pools are not detected and are disassembled as code, instead of being displayed as data.

- GCC does not generate call frame information. This means actions that rely on knowing the stack frame layout do not work. Specifically:
  - No stack backtrace is available. Only the current function is shown in the stack backtrace.
  - Step out does not work.

  Local variables and parameters are available in the variable view, however you must step over the function prologue code that sets up the stack frame before they show the correct values.

  Line number information is available, so the source view correctly displays the current source line.

IRQ and FIQ debugger internal variables

The ARM debuggers now support debug targets that create their own variables. These are named `$<proc_name>$<variable_name>`, where:

`<proc_name>` Is the name of the processor, as shown in the **Target** panel of the **Control** system view (for example, `ARM1020E`).

`<variable_name>`

Is the name of the variable, and can include:

`irq` (For example, `$ARM1020E$irq`.) A target can export this variable to provide a means of asserting the interrupt request pin. To trigger an interrupt manually, set the value to 1. To clear the interrupt, set it to 0. The processor CPSR must not be configured to disable interrupts.

`fiq` (For example, `$ARM1020E$fiq`.) A target can export this variable to provide a means of asserting the fast interrupt request pin. To trigger a fast interrupt manually, set the value to 1. To clear the fast interrupt, set it to 0. The processor CPSR must not be configured to disable fast interrupts.

Your debug target might create other variables (for example, `ARM1020E$other`). See the target documentation for details.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Code size improvements and improved optimization

ADS 1.1 optimization improvements have improved code density for compiled code over ADS 1.0.1. A number of additional optimizations have been introduced. In particular the compilers now reorder top-level data items when this will save space. For example, if the following global variables are defined:

```
char  a = 0x11;
short b = 0x2222;
char  c = 0x33;
int   d = 0x44444444;
```

the ADS 1.1 compilers optimizes the order and stores them in memory as:

```
char  a;
char  c;
short b;
int   d;
```

Impact

You cannot rely on the order of global variables in memory being the same as the order in which they are declared. If, for example, you have used a sequence of volatile global variables to map peripheral registers, you must rewrite your code to use structures.

Extensions to RDI support

AXD supports the following extensions to RDI 1.5.1:

- real-time extensions, required for RealMonitor (RDI 1.5.1rt)
- trace extensions required for Trace support (RDI 1.5.1tx)
- self-describing module extensions, required to support self-describing targets (RDI 1.5.1sdm).

Support for RealMonitor

RealMonitor is the ARM real-time debug solution. AXD has been enhanced to provide support for RealMonitor. AXD can now connect to a running target without halting the processor. RealMonitor requires the real-time extensions to RDI 1.5.1. In addition, the Gateway DLL now supports RealMonitor.

Support for self-describing modules

Self-describing modules are an extension to RDI 1.5.1 that enable a target to describe its capabilities and requirements to a debugger that supports RDI 1.5.1sdm. AXD supports RDI 1.5.1sdm, and can modify its interface to suit the target to which it is

Copyright © 1999, 2000 ARM Limited. All rights reserved.

connected. For example, the target can describe the number, name, and formatting requirements of its coprocessor registers to AXD, and AXD modifies its interface to represent the capabilities of the target.

Note

This means that AXD interface elements can change, depending on the target to which it is connected.

Changes to memory alignment

ADS 1.1 ensures that stack data is always 8-byte aligned. The new ATPCS requires that sp always points to 8-byte boundaries. Your assembly language code must preserve 8-byte alignment of the stack at all external interfaces.

In addition, the default implementations of `__user_initial_stackheap()`, `malloc()`, `calloc()`, `realloc()`, and `alloca()` now ensure that heap data is 8-byte aligned.

Impact

If you access stack data from assembly language you must ensure that you maintain 8-byte alignment of the stack at external interfaces.

If you have re-implemented the ARM C library default memory model, you must ensure that you maintain 8-byte alignment of the heap. In particular, you must ensure that your implementations of `__rt_heap_extend()`, `__user_heap_extend()` return 8-byte aligned blocks of memory. `__HeapProvideMemory()` is allowed to assume 8-byte alignment. It is recommended that your implementations of `__user_initial_stackheap()`, `__Heap_Alloc()` and `__Heap_Realloc()` maintain 8-byte alignment of heap memory.

If you use the `LDRD` or `STRD` instructions, you must ensure that the location accessed is 8-byte aligned. In ARM assembly language:

- you must set the alignment of any data section, or code section that contains data, using the `ALIGN` attribute to the `AREA` directive.
- you must use the `ALIGN` directive to ensure that data structures are 8-byte aligned.

For example:

```
    AREA example,CODE,ALIGN=3
    ;code
    ;code
my_struct  DATA
    ALIGN  8    ;aligned on 8 byte boundary
    DCB 1,2,3,4,5,6,7,8
```


Copyright © 1999, 2000 ARM Limited. All rights reserved.

The ADS 1.1 assembler supports two new directives to mark assembly units that contain functions that require, or preserve 8-byte alignment of the stack. This enables the linker to detect calls between code that does maintain 8-byte alignment, and code that does not maintain 8-byte alignment.

`PRESERVE8` Use this directive to mark assembly files that contain only functions that preserve 8-byte alignment of the stack.

`REQUIRE8` Use this directive to mark assembly files that contain at least one function that requires 8-byte alignment of the stack (for example, the stack is accessed with `LDRD`/`STRD` instructions.)

If you are using `LDRD`/`STRD` to access data objects defined in C or C++, you must use `__align(8)` to ensure that the data objects are properly aligned. `__align(8)` is a storage class modifier. This means that it can be used only on top-level objects. You can *not* use it on:

- types, such as typedefs, structure definitions
- function parameters.

It can be used in conjunction with **`extern`** and **`static`**. `__align(8)` only ensures that the object is 8-byte aligned. This means, for example, that you must explicitly pad structures if required.

Note

Output objects from a compilation or assembly are marked as requiring 8-byte alignment in the following circumstances:

- you specify the `REQUIRE8` directive, because you are using `LDRD` and `STRD` instructions in your assembly language code
- you allow the compiler to generate `LDRD` and `STRD` instructions by specifying the `-Oldrd` option
- you use the `__align(8)` qualifier to set the alignment of an object to an eight byte boundary.

These objects are unlikely to link with objects built with versions of ADS earlier than 1.1.

Angel moved to AFS

Angel is no longer shipped as part of ADS. Angel is now available as part of the ARM Firmware Suite.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.2.2 Differences in default behavior

This section describes how the default behavior of ADS 1.1 differs from that of ADS 1.0. The major changes are:

- The CodeWarrior IDE, fromELF, and armsd are now license-managed in the same way as other components of ADS.
- The linker now unmangles C++ symbol names when generating diagnostic messages or listings. This is the default. You can use the `-mangled` option to change the behavior.
- The byte order of an ARMulator target can now be set in the target configuration dialog in AXD and ADW.
- The ARM compilers now require the stack to be 8-byte aligned.
- AXD now starts up with much more of its state reproduced from the last session.
- The compilers now perform auto-inlining by default for optimization level `-O2`.
- The compilers now perform range-splitting optimization at optimization level `-O1` (previously only done for `-O2`).
- The default filename for binary output from the compiler is now `__image.axf`. That is, `armcc sourcename.c`, with no output name specified by the `-o` option, now generates `__image.axf`, not `sourcename`.
- The ARMulator default behavior has changed. Branches to address zero are now trapped only if you are running the FPE.

2.2.3 Changes to the compilers and libraries

This section describes:

- *New compiler options and pragmas* on page 2-11
- *Obsolete compiler options* on page 2-12
- *Changed behavior* on page 2-12
- *Changes to the inline assemblers* on page 2-14
- *Changes to the libraries* on page 2-14.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

New compiler options and pragmas

This section gives a brief summary of new compiler options for ADS 1.1. Refer to the Compilers chapter of the *ADS Compiler, Linker, and Utilities Guide* for detailed information. The following compiler options are new for ADS 1.1:

`-split_ldm` This option reduces the maximum number of registers transferred by `LDM` and `STM` instructions generated by the compiler.

`-O[no_]autoinline`

This option disables automatic inlining. It can be enabled with `-Oautoinline`.

`-O[no]ldrd` This option controls optimizations specific to ARM Architecture v5TE processors. The default is `-Ono_ldrd`. If you select `-Oldrd`, and select an Architecture v5TE `-cpu` option such as `-cpu xscale`, the compiler:

- Generates `LDRD` and `STRD` instructions where appropriate.
- Sets the natural alignment of **`double`** and **`long long`** variables to eight. This is equivalent to specifying `__align(8)` for each variable.

Note

If you select this option, the output object is marked as requiring 8-byte alignment. This means that it is unlikely to link with objects built with versions of ADS earlier than 1.1.

`-auto_float_constants`

This option changes the default handling of unsuffixed floating-point constants.

`-Wk` This option turns off warnings that are given when the `-auto_float_constants` option is used.

`-O[no_]known_library`

The `-Oknown_library` option enables specific optimizations that are dependent on the supplied ARM C library. `-Ono_known_library` is the default, and is required if you re-implement any part of the C library.

`-fpu softvfp+vfp`

This option selects a floating-point library with pure-endian doubles and software floating-point linkage that uses the VFP hardware. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFP unit.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

`-fpu vfpv1` Selects hardware Vector Floating Point unit conforming to architecture VFPv1. This option is a synonym for `-fpu vfp`. It is not available for Thumb.

`-fpu vfpv2` Selects hardware Vector Floating Point unit conforming to architecture VFPv2. This option is not available for Thumb.

`#pragma import(`*`symbol_name`*`)`

This pragma generates an importing reference to *`symbol_name`*. This is the same as the assembler directive:

```
IMPORT symbol_name
```

New predefined macros

The following predefined macros are new for ADS 1.1:

`__TARGET_FEATURE_DOUBLEWORD`

Set if the target architecture supports the `PLD`, `LDRD`, `STRD`, `MCRR`, and `MRRC` instructions.

`__TARGET_FPU_SOFTVFP_VFP`

Set by the `-fpu softvfp+vfp` option.

Obsolete compiler options

The following compiler option, deprecated in ADS 1.0, is not supported in ADS 1.1:

`-dwarf1` This option specifies DWARF1 debug table format. Specify `-dwarf2` instead of `-dwarf1`.

Impact

You must modify existing makefiles that use this option.

Changed behavior

ADS 1.1 introduces the following changes to the behavior of the compilers:

- The default filename for binary output from the compiler is now `__image.axf`. That is, `armcc sourcename.c`, with no output name specified by the `-o` option, now generates `__image.axf`, not `sourcename`.
- Range splitting optimizations are turned on for `-O1`.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

- Output from the `-S` and `-S -fs` options now displays standard register names, such as r0-r3, instead of their ATPCS equivalents (a1-a4). The output from `-S -fs` is now written to *filename*.txt, instead of *filename*.s.
- When compiling with `-zo`, output sections now use the same name as the function that generates the section. For example:

  ```
  int f(int x) { return x+1; }
  ```

  compiled with `-zo` gives:

  ```
          AREA ||i.f||, CODE, READONLY
  f PROC
          ADD      r0,r0,#1
          MOV       pc,lr
  ```

- When compiling for ARM architecture v5TE, the compilers now use the one-cycle 16-bit multiply instruction when multiplying two16-bit (short) operands to produce a single 32-bit result.
- The compilers now support out of order inlining. This means you can call an inline function before it is defined. In accordance with standard C, you must still declare the function prototype before it is called.
- Additional optimizations have been introduced. The compilers do not generate:
  - Unused static functions for optimization levels `-O1` and `-O2`.
  - Unused inline functions for all optimization levels. (This is unchanged behavior.)
  - Unused static RW data for `-O1` and above.
  - Unused static const data for all optimization levels.

In addition:

- The compilers now reorder top-level data items when this will save space. For example, if the following global variables are defined:

  ```
  char  a = 0x11;
  short b = 0x2222;
  char  c = 0x33;
  int   d = 0x44444444;
  ```

  the ADS 1.1 compilers optimizes the order and stores them in memory as:

  ```
  char  a;
  char  c;
  short b;
  int   d;
  ```

- the compilers now place zero initialized global and static definitions such as:

  ```
  int a=0;
  ```


Copyright © 1999, 2000 ARM Limited. All rights reserved.

in the ZI data area. The variables are initialized by the C libraries at run time. Previously such variables were placed in the RW area.

Changes to the inline assemblers

The inline assemblers support the ARMv5TE instructions. The inline assemblers do not support VFP floating-point instructions.

Changes to the libraries

The ARM C libraries are now compiled with the `-split_ldm` compiler option. The libraries preserve 8-byte alignment of the stack, and the default memory model functions preserve 8-byte alignment of the heap. In addition:

- floating-point exceptions are disabled by default
- the ANSI C and C++ run-time support libraries are now combined
- a number of additional minor library variants are provided.

2.2.4 Changes to the assembler

This section describes:

- *New instructions and directives* on page 2-14
- *New assembler options* on page 2-15
- *Obsolete assembler options* on page 2-15
- *Changed behavior* on page 2-16.

New instructions and directives

The assembler provides support for new ARM architecture v5TE processors, including the Intel XScale. The instruction set is now documented in the new *ADS Assembler Guide*.

The ADS 1.1 assembler provides the following new directives:

`PRESERVE8` Use this directive to mark assembly files that contain only functions that preserve 8-byte alignment of the stack.

`REQUIRE8` Use this directive to mark assembly files that contain at least one function that requires 8-byte alignment of the stack

The assembler also supports the XScale coprocessor instructions `MAR`, `MRA`, `MIA`, `MIAPH`, and `MIAxy`

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

New assembler options

This section gives a brief summary of new assembler options for ADS 1.1. Refer to the the *ADS Assembler Guide* for detailed information. The following assembler options are new for ADS 1.1:

`-fpu softvfp+vfp`

This option selects software floating-point library with pure-endian doubles, software floating-point linkage, and requiring a VFP unit. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFP unit.

`-fpu vfpv1` Selects hardware Vector Floating Point unit conforming to architecture VFPv1. This option is a synonym for `-fpu vfp`. It is not available for Thumb.

`-fpu vfpv2` Selects hardware Vector Floating Point unit conforming to architecture VFPv2. This option is not available for Thumb.

`-split_ldm` This option instructs the assembler to fault `LDM` and `STM` instructions if the maximum number of registers transferred exceeds:

- five, for all `STM`s, and for `LDM`s that do not load the PC
- four, for `LDM`s that load the PC.

New predefined register names

ADS 1.1 predefines the following floating-point register names:

- s0-s31
- S0-S31
- d0-d15
- D0-D15.

Impact

You cannot use these names as user-defined label or symbol names in your assembly language code.

Obsolete assembler options

The following assembler options, deprecated in ADS 1.0, are not supported in ADS 1.1:

`-dwarf1 and -dwarf`

This option specifies DWARF1 debug table format. Specify `-dwarf2` instead of `-dwarf1`. `-dwarf` was a synonym for `-dwarf1`.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Impact

You must modify existing makefiles that use these options.

Changed behavior

ADS 1.1 introduces the following changes to the behavior of the assembler:

- The assembler now faults a call to a `GET` directive from within a macro.
- The assembler now faults the use of built-in variable names or predefined symbol names as a user symbol, such as a macro name. In ADS 1.0 the assembler silently ignored such usage.
- The assembler is now much stricter and more consistent in faulting usage that does not conform to the ARM Architecture Reference manual. For example:

```
    CMP     ip,a3,ASL #0
; Generates Warning: A1484E: Obsolete shift name 'ASL', use LSL instead
```

  and:

```
    SWP r0,r1,[r0]
; Generates Warning: A1477W: This register combination results
; in UNPREDICTABLE behavior
```

2.2.5 Changes to the linker

This section describes:

- *New linker options* on page 2-16
- *Changed linker behavior* on page 2-17.

New linker options

This section gives a brief summary of new linker options for ADS 1.1. Refer to the *ADS Compiler, Linker, and Utilities Guide* for detailed information. The following linker options are new for ADS 1.1:

`-callgraph` This option creates a static callgraph of functions in HTML format. The callgraph gives stack usage, definition, and reference information for all functions in the image.

`-edit file` This option enables you to specify a *steering file* containing commands to edit the symbol tables in the output binary.

`-unmangled` This option instructs the linker to display unmangled C++ symbol names in diagnostic messages and listings.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

`-mangled` This option instructs the linker to display mangled C++ symbol names in diagnostic messages and listings.

New scatter loading attributes

The scatter load syntax has been extended to include a new attribute for execution regions:

`FIXED` Fixed address. Both the load address and execution address of the region is specified by the base address (the region is a root region.)

Changed linker behavior

The following changes have been made to the linker behavior:

- The linker now unmangles C++ symbol names by default, in all listings and diagnostic messages.
- The linker generates two new region-related symbols:

  `Image$$`*`region_name`*`$$Limit`

  Address of the byte beyond the end of the execution region.

  `Image$$`*`region_name`*`$$ZI$$Limit`

  Address of the byte beyond the end of the ZI output section in the execution region.

  See the description of linker-defined symbols in the *ADS Compiler, Linker, and Utilities Guide* for more information, including a description of how to use the `Image$$`*`region_name`*`$$ZI$$Limit` symbol to place a heap directly after the ZI region. For new projects it is recommended that you use the region-related symbols rather than section-related symbols.
- The linker no longer generates a warning message if there is a duplicate definition of a symbol:

  `Both ARM & Thumb versions of symbol present in image`
- The linker options `-split` and `-rwpi` now assume `-rw-base 0` if no `-rw-base` value is specified.

2.2.6 Changes to fromELF

This section describes:

- *New fromELF options* on page 2-18
- *Changed behavior* on page 2-18.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

New fromELF options

This section gives a brief summary of new fromELF options for ADS 1.1. Refer to the *ADS Compiler, Linker, and Utilities Guide* for detailed information. The following fromELF options are new for ADS 1.1:

`-vhx` This option outputs Verilog Hex Format.

`-base n` This option specifies the base address of the output for Motorola S-record (`-m32`), Intel Hex (`-i32`), and Extended Intellec Hex (`-ihf`) hex formats.

memory_config

This option outputs multiple files for multiple memory banks. This option is available only if `-vhx` or `-bin` is specified as the output format.

Changed behavior

The following changes have been made to fromELF:

- fromELF can now disassemble ARMv5TE instructions.
- fromELF is now license-managed through FLEX*lm*
- fromELF now issues a warning for the `-aif` and `-aifbin` output options. These formats are no longer supported.
- the fromELF -S option now prints size information.

2.2.7 Changes to the Flash downloader

The default `flash.li` Flash download utility is now targeted at the ARM Integrator board. Source code for the Integrator `flash.li` is not supplied.

Source code for the ARM Development (PID) board is supplied in *install_directory*`\Examples\flashload`.

2.2.8 Changes to the debuggers

This section describes:

- *Changes to AXD* on page 2-18
- *Changes to armsd* on page 2-20
- *Changes to ADW and ADU* on page 2-20.

Changes to AXD

The AXD interface has been significantly enhanced, including:

- AXD disassembles ARMv5TE instructions.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

- AXD displays the XScale coprocessors CP0, CP13, CP14, and CP15 in appropriate formats.
- AXD data display has been enhanced to allow display in a choice of many different formats.
- AXD now displays breakpoints and watchpoints in separate lists.
- AXD persistence has been improved so that more of the previous GUI state can be restored at the start of a new session.
- AXD can load standard ELF/DWARF2 images produced by the GNU toolchain. See *Improved support for debugging third party images* on page 2-5 for more information.
- AXD adds support for RealMonitor. AXD can now connect to a running target without stopping the processor.
- The debug view is improved. See *Improved debug view* on page 2-4 for more information.
- AXD supports RDI *self-describing modules*. This means that AXD can reconfigure itself to suit the capabilities of the target if the target supports self-describing modules.
- AXD now allows targets to export their own debugger internal variables. See *IRQ and FIQ debugger internal variables* on page 2-6 for more information.

The AXD command-line interface has been improved. This has introduced some incompatibilities with ADS 1.0.1:

- Inputbase as a CLI property has been removed from both the properties dialog and from the `format` command. In addition, you must use a prefix to specify a nondecimal format:

  `0x/0X` specifies a hexadecimal value.

  `o/O` specifies an octal value.

  `b/B` specifies a binary value

  No prefix specifies a decimal value.
- The `asd` command is replaced by the `sdir` (source directory) and `ssd` (set source directory) commands.
- The `format` command has changed. The parameters have new meanings, and `format` is no longer used to set up current input base from the command line. If the old style `format` command is used, an error message is displayed.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Changes to armsd

The armsd debugger has been enhanced in the following ways:

- armsd disassembles ARMv5TE instructions.
- armsd is now license-managed through FLEX*lm*.
- armsd can set and display the 40-bit XScale CP0 register, in a similar way to current ARM usage. The following example shows how to use armsd to write the 40-bit value `0x9876543210` to register CP0, and read CP0 again:

  ```
  armsd: cw 0 0 0x76543210 0x98
  armsd: cr 0
  c0 = 0x76543210 FFFFFF98
  ```

  armsd reads the register as two 32-bit words, and sign extends bit 39 into the upper 24 bits.
- armsd now accepts `-cpu [`*`name`*`] list` to list the available processors in a target:

  `armsd -cpu list`
  : lists available processors of standard targets (ARMulator and Remote_A).

  `armsd -armul -cpu list`
  : lists available processors of ARMulator.

  `armsd -target `*`dllname`*` -cpu list`
  : lists available processors of the specified target.
- armsd on UNIX loads its RDI targets dynamically.
- the armsd `return` command is no longer supported.

Obsolete armsd options

The `-proc` option is obsolete. Use `-cpu` instead.

Changes to ADW and ADU

ADW and ADU now disassemble ARMv5TE instructions.

2.2.9 Changes to ARMulator

This section describes changes to the ARMulator, including:

- *License management* on page 2-21
- *Integrated ARMulator and new processor models* on page 2-21
- *New API for memory models* on page 2-21

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

- *New configuration mechanism* on page 2-21
- *ARMulator byte order set from the debuggers* on page 2-22
- *Changes to default behavior* on page 2-22.

License management

The ARMulator is now license-managed at the model level, through FLEX*lm*.

Integrated ARMulator and new processor models

The ARMulator has been restructured to provide a single interface to all processor models that is easier to use and modify. All ARMulator models are accessible through a single target DLL (`armulate.dll`). The BATS DLL is not supplied with ADS 1.1. The ARMulator is now supplied as a shared object under UNIX.

The ARMulator has been upgraded to support the latest processors. It provides new models of:

- the ARM9 (ARM946E and ARM966E)
- ARM10, including VFP10
- XScale.

New API for memory models

The ARMulator provides a new memory model interface that enables you to add memory and peripheral models without rebuilding the complete ARMulator. The ARMulator rebuild kit has been replaced with the ARMulator Extension Kit.

New configuration mechanism

The `armul.cnf` configuration mechanism has been split to separate features that can be edited and those that cannot. The ARMulator recognizes two file extensions:

- Files ending `.dsc`, such as `armulate.dsc`, are supplied with ARMulator DLLs. They describe the cores and peripherals the DLL can emulate. They are not intended to be edited.

- Files ending `.ami` are intended to be edited. A `.ami` file can define one or more named systems, for selection by the `-cpu` option in armsd or a dialog box in a GUI debugger.

The `armulate.dll` model looks at environment variables `ARMDLL` and `ARMCONF` for paths to search for its configuration files. It loads all the files with those extensions it can find (on the paths specified by the `ARMDLL` and `ARMCONF` environment variables). If it cannot find any, it issues an error message and fails to initialize.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

In order to avoid loading files that are not meant for the `armulate.dll` product, it examines each file and checks that it starts with:

```
;; ARMulator configuration file type 3
```

ARMulator byte order set from the debuggers

The ARMulator configuration dialog can now set the byte order of the simulated processor from within the debugger. Also, there are additional radio buttons in the configuration dialog to set the default startup behavior for ARMulator models that have a CP 15. See the *ADS Debuggers Guide* for detailed information.

Changes to default behavior

The ARMulator default behavior has changed. Branches to address zero are now trapped only if you are running the FPE.

2.2.10 Changes to the CodeWarrior IDE

The CodeWarrior IDE is now license-managed using FLEX*lm*.

CodeWarrior IDE project files are stored in a slightly different format to those of ADS 1.0.1. The CodeWarrior IDE automatically converts existing project files to the new format and displays a dialogue box to inform you.

CodeWarrior configuration dialogs have been updated to support the changed tool options.

The ARM Features configuration panel has been added to support license-restricted license-managed features.

On Windows 95 it is recommended that you have a minimum patch level of OSR_2, or that you have Internet Explorer 5 installed, to ensure that you have the latest OLE DLLs.

2.2.11 Changes to the examples

The example code supplied with ADS in `install_directory\Examples` has been supplemented with the following additional examples:

- The `rom_integrator` directory contains a version of the `rom` example that is targeted at the ARM Integrator board.
- The `mmugen` directory contains the source and documentation for the MMUgen utility. This utility can generate MMU pagetable data from a rules file that describes the virtual to physical address translation required.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.3 Changes between ADS 1.0 and SDT 2.50/2.51

This section describes the changes between ADS 1.0 and SDT 2.50/2.51. It contains the following subsections:

- *Functionality enhancements and new functionality* on page 2-23
- *Differences in default behavior* on page 2-33
- *Changed compiler behavior* on page 2-38
- *Changed assembler behavior* on page 2-44
- *Changed linker behavior* on page 2-47
- *Obsolete components and standards* on page 2-48.

2.3.1 Functionality enhancements and new functionality

The ADS 1.0 release of the ADS introduced numerous enhancements and new features. The major changes are described in:

- *Support for new processors (ARM9E and ARM10)*
- *New ARM/Thumb procedure call standard* on page 2-24
- *Floating-point support* on page 2-24
- *Byte order of long long and double* on page 2-25
- *Remote Debug Interface* on page 2-26
- *Debuggers* on page 2-26
- *ARMulator* on page 2-27
- *Angel and Remote_A* on page 2-27
- *Libraries* on page 2-28
- *Library manager* on page 2-28
- *CodeWarrior IDE* on page 2-29
- *Linker* on page 2-29
- *Compilers* on page 2-30
- *Assembler* on page 2-31
- *License management* on page 2-32.

Support for new processors (ARM9E and ARM10)

ADS introduces support for the new ARM9E and ARM10 processors.

The new ARM9E instructions are supported by the assembler, the inline assembler of the C and C++ compilers, the debuggers, and the ARMulator.

The new ARM10 instructions are supported by the assembler, the inline assembler of the C and C++ compilers, the debuggers, and the *Basic ARM Ten System* (BATS) ARMulator model.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Note

BATS is no longer shipped with ADS 1.1.

The compiler performs instruction scheduling for ARM10 code by re-ordering machine instructions to gain maximum speed and minimize wait states. The linker uses `BLX` in interworking veneers when the underlying architecture (the ARM9E and ARM10, for example, have architecture 5) supports it.

New ARM/Thumb procedure call standard

The Procedure Call Standard has been redesigned to:

- give equal emphasis to ARM and Thumb
- interwork between ARM-state and Thumb-state for all variants
- reduce the number of variants
- support position-independence
- produce compact code (especially with Thumb)
- be binary compatible with the previous most commonly used APCS variant.

The new *ARM/Thumb Procedure Call Standard* (ATPCS) enables a consistent ARM and Thumb definition of Read Only Position Independence (also called Position Independent Code), and Read Write Position Independence (also called Position Independent Data) for both ARM and Thumb.

Floating-point support

Enhanced floating-point support is available in the compiler, assembler, and debugger:

- The compiler, assembler, and debugger support the new VFP floating-point architecture in scalar mode.
- The compiler can generate VFP instructions for **`double`** and **`float`** operations. (The inline assembler, however, does not support VFP.)
- The assembler supports VFP in vector mode. (New register names and directives are available.)
- The compiler and assembler command-line option `-fpu` specifies the FPA hardware, VFP hardware, or software variants.

  Choose `-fpu FPA` or `-fpu softFPA` to retain the old SDT 2.50/2.51 format.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Note

The order of the words in a little-endian **double** is different for FPA and VFP. If you select `-fpu FPA` or `-fpu softFPA` the SDT 2.50/2.51 format is used. If you select `-fpu VFP` or `-fpu softVFP` the new format is used.

There is no functional difference between SoftFPA and SoftVFP. Both implement IEEE floating-point arithmetic by subroutine call, and both use the IEEE encoding of floating-point values into 32-bit words. However, the ordering of the two halves of a **double** is different for little-endian code. See *Byte order of long long and double* for details.

Byte order of long long and double

The compilers and assembler now support industry-standard **long long** and **double** types in both little-endian and big-endian formats. In SDT 2.50/2.51, the formats of little-endian **double** and big-endian **long long** are nonstandard.

If a big-endian 64-bit quantity is represented as `abcdefgh`, with `a` being the most significant byte and `h` the least significant byte, the standard little-endian format is `hgfedcba`. SDT 2.50/2.51 uses the following nonstandard formats:

`efghabcd` For big-endian **long long**.

`dcbahgfe` For little-endian **double**.

Impact

The format of **long long** is always industry-standard in ADS 1.0. There is no impact if you have used little-endian **long long**. If you previously used big-endian **long long**, you must recompile your code and ensure that it conforms to the new format.

There is no impact if you have used big-endian **double**. If you previously used little-endian **double** and hardware floating-point (FPA), you must continue to use the old little-endian **double** format and select the `-fpu fpa` option in ADS.

If you previously used little-endian double and software floating-point, you can choose whether or not to change to the new format:

- Use `-fpu softFPA` or `-fpu FPA` to retain the old format.
- Use `-fpu softVFP` or `-fpu VFP` to use the industry-standard format. You must recompile code that defines or references **double** types.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Remote Debug Interface

A new variant of the *Remote Debug Interface* (RDI 1.5.1) is introduced in ADS. The version used in SDT 2.50/2.51 was 1.5.

The ADW debugger has been modified to function with RDI 1.0, RDI 1.5, or RDI 1.5.1 client DLLs. AXD works with RDI 1.5.1 targets only.

Debug targets that are released as part of ADS (ARMulators, Remote_A, and Gateway) have been upgraded to RDI 1.5.1.

Impact

Third-party DLLs written to use RDI 1.5 will continue to work with the versions of ADW and armsd shipped with ADS, but will only work with AXD if the DLL is, and reports itself as, RDI 1.5.1 capable. Third-party debuggers will fail to work with the ADS ARMulators, Remote_A, and Gateway DLLs unless the debuggers conform to RDI 1.5.1.

Debuggers

A new debugger, AXD, is available for use on Windows or UNIX in addition to the existing ADW and ADU. ADW has been enhanced.

All debug agents and targets in ADS support RDI 1.51, a new version of the Remote Debug Interface. The debuggers support all the debug agents (for example ARMulator and Remote_A) that are released as part of ADS. In addition, all debuggers except armsd support Multi-ICE 1.4:

- ADW supports all ADS debug agents, Multi-ICE 1.3, and Multi-ICE 1.4
- ADU supports all ADS debug agents, and Multi-ICE 1.4
- Armsd supports all ADS debug agents
- AXD supports all ADS debug agents and Multi-ICE 1.4.

AXD

The new debugger provides a modern GUI with improved window management, data display, and data manipulation. The debugging views have been redesigned to make the display more relevant to the data. This includes in-place expansion, in-place editing and validation, data sensitive formatting and editing, coloring modified data, and greater user control over formatting and structure.

ADW

ADW enhancements are:

- Support for VFP floating-point opcodes and registers.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

- Improved stack-unwinding due to the use of DWARF2 descriptions. In ADS, all standard library functions carry DWARF frame unwinding descriptions with them. These are always generated by ADS compilers and there is new assembler support in ADS to facilitate their generation for hand-written assembly language.

Impact

AXD can debug RDI 1.5.1 targets only. All ARM-supplied debug targets (Multi-ICE, ARMulator, Remote_A, and gateway) support RDI 1.5.1. For non-ARM debug targets that support RDI 1.5 or RDI 1.0, use ADW instead of AXD.

There is no support for conversion of ADW *obey* files to AXD scripts. If existing obey files are important, use ADW instead.

ARMulator

The ARMulator has been enhanced to support RPS Interrupt Controller and Timer peripheral models (as defined in ARM DDI 0062D). The ARMulator supports the following new processor models:

- ARM9E
- ARM10T™
- ARM1020T™.

The ARM10 models do not support VFP.

There is also a new stack usage monitor memory model available for all processor models except ARM10T and ARM1020T.

The ARMulator supports RDI 1.5.1.

Angel and Remote_A

Angel and Remote_A enhancements are:

- Remote_A connection supports RDI 1.5.1.
- Improved reliability when semihosting.
- Additional Angel ports and improved integration with uHAL.
- Improved coprocessor support, for example FPA (ARM7500) and VFP (ARM10) coprocessors.
- Support for dynamically loaded hardware drivers for the host on Windows and UNIX.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Hardware other than serial, parallel, or ethernet ports can be used to communicate with Angel. The GUI interface for Remote_A is extended into the loaded driver.

Libraries

All Libraries (C, C++, math, and floating-point) are released as a set of object code variants that cover all possible choices of Procedure Call Standard and all processor architecture versions. A limited set of variants is required because the libraries have been restructured to remove the necessity for some combinations. The compilation and linking system has been re-engineered so that the correct library variants are automatically linked in for the chosen compilation options. The linker is able to identify the correct library variant from attributes embedded in the ELF. This re-engineering makes the library variants much easier to use and removes the requirement to rebuild different variants.

The C library has been improved and restructured so that there is no requirement for a separate embedded C library. The C library chapter in the *ADS Compiler, Linker, and Utilities Guide* describes in detail how to construct target-specific libraries.

New real-time (near constant time) versions of the heap management functions `malloc()`, `free()`, `realloc()`, and `calloc()` are provided.

The floating-point libraries have improved performance and functionality. Two versions are provided:

- The version identified by the files beginning with `f_` conforms to IEEE 754 accuracy standards and meets the floating-point arithmetic requirements of the C and Java language standards.
- The version identified by the files beginning with `g_` provides selectable IEEE rounding modes and full control of IEEE exceptions, but at some performance cost.

The Math library has better accuracy and a wider variety of functions (for example, gamma function, cube root, inverse hyperbolic functions).

Library manager

The library manager is armar. The ARM librarian enables sets of ELF format object files to be collected together and maintained in libraries. You can pass such a library to the linker in place of several ELF files. armar files are compatible with the UNIX archive format ar.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Impact

The linker supports the deprecated ALF library format. Use armar for new libraries and migrate your existing libraries to armar.

CodeWarrior IDE

ARM has licensed the CodeWarrior IDE from Metrowerks and is making this available within ADS. This replaces APM on Windows platforms. (It is not available on UNIX platforms).

The CodeWarrior IDE provides a simple, versatile, graphical user interface for managing your software development projects. You can use CodeWarrior for the ARM Developer Suite to develop C, C++, and ARM assembly language code targeted at ARM processors. The CodeWarrior IDE enables you to configure the ARM tools to compile, assemble, and link your project code.

CodeWarrior IDE configuration dialogs

The CodeWarrior IDE dialog boxes are used to select the new features available in the compilers, assembler, and the linker.

Each selectable option on the dialog boxes has a tool tip that displays the command-line equivalent for the option.

Impact

Existing APM projects are not usable with CodeWarrior. There is no support for conversion of `.apj` files to CodeWarrior projects. Use the CodeWarrior IDE for new projects. Migrate your existing APM projects to use the CodeWarrior IDE.

Check the assembler, compiler, and linker options for your new or migrated projects as the defaults for ADS 1.0 are different from the defaults for the SDT 2.50/2.51.

Linker

The major linker enhancements are:

- Support for ELF object code.
- Support for automatic selection of the correct library variant.
- Improved scatter-loading features to support new execution region attributes:
  - Position Independent (PI)
  - Relocatable (RELOC)
  - linked at a fixed address (ABSOLUTE)

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

— simple Overlay (OVERLAY).

- Direct support for ROPI and RWPI procedure call standard variants.
- Support for outputting symbol definitions from a link step and reading them in a later link step (support for system code at a fixed address).

Impact

Update your projects or makefiles to link with the appropriate options. In most cases you will not have to change your source code to use the new options.

Check the assembler, compiler, and linker options for your new or migrated projects as the defaults for ADS 1.0 are different from the defaults for armlink in SDT 2.50/2.51.

See *Changed linker behavior* on page 2-47 and the *ADS Compiler, Linker, and Utilities Guide* for more information.

Compilers

Extensive improvements have been made to the compilers.

C compilers

The following improvements have been made to the C Compiler:

- Assembly language output generated with the `-S` option to the ARM and Thumb compilers can now more easily be assembled. The compilers add `ASSERT` directives for command-line options such as ATPCS variants and byte order to ensure that compatible compiler and assembler options are used when reassembling the output.
- The inline assembler supports the new ARM9E and ARM10 instructions.
- Instruction scheduling for ARM10 minimizes wait states.
- the new VFP architecture is supported.

New compiler options are provided for:

- controlling warnings
- selecting optimization
- generating position-independent code and position-independent data.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

C++ compilers

The C++ compilers included with ADS inherit all the benefits of the C compiler. The following additional improvements have been introduced since C++ version 1.10:

- Rogue Wave Library 2.01. This includes the Rogue Wave iostream implementation. The iostream implementation supplied with C++ version 1.10 has been removed. Replace references to `stream.h` and `iostream.h` with `iostream`.
- Support for the EC++ informal standard.
- Updated vtables to support ROPI.
- Improved template handling.

In addition, improvements have been made to the C++ compilers syntax and semantic checking in both strict and non-strict modes. If previously successful programs now fail to compile, please check their syntax first, before concluding that there is a compiler fault.

Other general improvements are support for:

- **`mutable`**
- **`explicit`**
- covariant return types for left-most inheritance
- pseudo-destructors
- aggregates with allow complicated initializations
- template classes with static data members
- temporary destruction order for arguments to functions
- **`explicit`** casts to **`private`** bases
- inline functions
- better overload resolution
- declarations in conditional statements.

See *Changed compiler behavior* on page 2-38 and the *ADS Compiler, Linker, and Utilities Guide* for more information.

Assembler

Enhancements to the assembler include:

- the assembler provides support for the latest ARM processors
- the assembler outputs ELF object code.

There are considerable changes to assembler directives. See *Changed assembler behavior* on page 2-44 and the *ADS Compiler, Linker, and Utilities Guide* for more information.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

License management

ADS components are license-managed by FLEX*lm*. See the *ADS Installation and License Management Guide* for more information.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.3.2 Differences in default behavior

The differences in the default behavior of ADS compared to SDT 2.50/2.51 are described in:

- *Object and library compatibility* on page 2-33
- *Entry point used with debugger* on page 2-34
- *Entry point set by linker option* on page 2-34
- *ADW* on page 2-34
- *ARMulator* on page 2-35
- *ELF, AIF, Binary AIF, IHF and Plain Binary Image formats* on page 2-35
- *Floating-point exceptions* on page 2-35
- *Stack unwinding* on page 2-36
- *Source directory variable in armsd and ADW* on page 2-36.

Object and library compatibility

As a consequence of the new features introduced with ADS, ADS object files and libraries are not guaranteed to be compatible with SDT 2.50/2.51 object files and libraries. You can link SDT 2.50/2.51 objects and libraries with ADS images, but you must ensure that your objects are built with appropriate procedure call standard options, and that the following restrictions are observed:

- You must choose the SDT 2.50/2.51 default Procedure Call Standard options when using SDT 2.50/2.51 (`/hardfp` excluded), and the ADS 1.0 default Procedure Call Standard options when using ADS.

- In ADS, you must use `-fpu FPA`, `-fpu softFPA`, or `-fpu none`. You cannot use the default option of `-fpu softVFP`.

- The format of big-endian **`long long`** has changed. This means that there is no compatibility between ADS and SDT big-endian code if you use **`long long`**.

- There is no equivalent in ADS to the SDT `-apcs /nofpregargs` option for functions that return a floating-point value. Functions that are built with the `-apcs /nofpregargs` option, but do not return a floating-point value, are compatible with functions declared using the new `__softfp` keyword.

- An SDT 2.50/2.51 object and an ADS object will be incompatible if they map the same datum using a **`struct`** type T, whether through use of T * pointers or **`extern`** T, and T contains only fields of short alignment.

  A field has short alignment if its type is:

  — `[`**`unsigned`**`]` **`short`** `[`**`array`**`]`

  — `[`**`unsigned`**`]` **`char`** `[`**`array`**`]`

Copyright © 1999, 2000 ARM Limited. All rights reserved.

— a `short enum` type
— a `struct` containing only fields of short alignment.

In addition, if you link with SDT 2.50/2.51 objects you cannot take advantage of some ADS debug enhancements. In particular, you cannot unwind the stack through SDT 2.50/2.51 code.

Entry point used with debugger

When an image with an entry point is loaded:

- the CPSR register is set to the value corresponding to a warm boot
- the IRQ and FIQ flags are set (disabling all interrupts)
- mode is set to Supervisor
- Condition Code flags are unchanged
- the processor executes in ARM state.

If the image contains no entry point, no change is made to the CPSR.

Default interrupt settings for debug targets

The ADS default for interrupt settings is different to that for SDT, and more accurately reflects the hardware power-up settings of an ARM core. The debuggers no longer enable interrupts during start-up. Interrupts are initially disabled for all debug targets except Angel (which requires interrupts to be turned on in order to behave correctly).

In SDT 2.50/2.51, initially `cpsr = %ift_SVC32` for all targets.

In ADS, initially `cpsr = %IFt_SVC` for all targets except Angel, which has `%ift_SVC`.

Entry point set by linker option

The `-entry` option sets the entry point for an image. There can be only one instance of the `-entry` option to the linker.

Impact

Multiple `-entry` settings generate an error.

ADW

ADW now defaults to VFP mode for the display of floating-point and double values, and for floating-point registers.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

The SDT 2.50/2.51 version of ADW allowed some debug target settings to be configured by the debugger tab on the configuration screen (for example, ARMulator memory maps and byte order). For ADS debug targets, this tab is greyed out and the configuration button must be used instead. This button invokes the configuration box in the RDI Target.

Impact

The RDI Target configuration box has been extended to handle memory maps and byte order. The debugger tab is still available for old debug target DLLs.

ARMulator

FPE is now deselected by default in ARMulator. If you need FPE support for armsd, use the command-line option `-FPE`. If you need FPE support for ADW or AXD, select the **FPE** option in the ARMulator configuration dialog.

Map file selection has changed since SDT 2.50/2.51. The local/global/none map file selection dialog has been replaced with a single map file selection.

ELF, AIF, Binary AIF, IHF and Plain Binary Image formats

The default, and only supported, image format is ELF.

Impact

The preferred way to generate an image in a non-ELF image (such as plain binary or AIF) is to use the fromELF tool to translate the ELF image into the required format.

Floating-point exceptions

The ADS tools have been changed to conform to the IEEE specification. The SDT2.50/2.51 tools set the default response to floating-point Invalid-Operation, Divide-By-Zero and Overflow to be a trap causing program termination. This is contrary to IEEE 754 section 7, that states that “The default response to an exception shall be to proceed without a trap.”

Impact

To restore exception handling to the SDT 2.50/2.51 default, make the call shown in Example 2-1 before using any floating-point operations. The call should preferably be at the beginning of `main()`.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Example 2-1

```
#include <fenv.h>
#define EXCEPTIONS (FE_IEEE_MASK_INVALID | FE_IEEE_MASK_DIVBYZERO | \
       FE_IEEE_MASK_OVERFLOW)
__ieee_status(EXCEPTIONS, EXCEPTIONS);
```

Stack unwinding

The compilers now always generate DWARF2 stack-unwinding descriptions. In SDT 2.50/2.51 they were only generated if the `-g` option was specified (for debug information). The assembler generates stack-unwinding descriptions if the new frame directives are used. The debuggers rely on the stack-unwinding descriptions for stack backtrace.

Impact

If you want to unwind stacks when debugging assembler code, ensure that you use the new frame directives. Stack-unwinding descriptions are automatically generated by the ADS compilers and are included in the libraries released with ADS, so you have to change only assembly language code and legacy SDT2.50/2.51 code not compiled with debug information (`-g` option). You can examine disassembled output from the compilers to see how to use the assembler frame directives correctly.

Source directory variable in armsd and ADW

The `$sourcedir` variable used by armsd and ADW defaults to NULL if no value is specified. In addition, the delimiter used to separate multiple pathnames has been changed from a space to a semicolon.

The variable is used only to specify alternative search paths to the debuggers. You must use the following conventions when specifying search paths:

- Enclose the full pathname in double quotes.
- In ADW and armsd under Windows DOS, escape the backslash directory separator with another backslash character. For example:

  `"c:\\mysource\\src1"`
- Separate multiple pathnames with a semicolon, not with a space character. For example:

  `"c:\\mysource\\src1;c:\\mysource\\src2"`

  You can also specify long pathnames containing space characters. For example:

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

```
"c:\\my source\\src1;c:\\my source\\src2"
```


Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.3.3 Changed compiler behavior

This section describes compiler behavior that is new, changed, deprecated, or obsolete. Obsolete features are identified explicitly. Their use is faulted in ADS. Deprecated features will be made obsolete in future releases. Their use is warned about in ADS.

New compiler options

The following new warning options are available in the compilers:

-We Turn off warnings about pointer casts

-Wm Turn off warnings about multi-character **char** constants

-Wo Turn off warnings about implicit conversion to signed **long long**

-Wq Turn off warnings about C++ constructor initialization order

-Wy Turn off warnings about deprecated features.

Use -W+*option* to turn a warning on. For example use -W+e to turn on warnings about pointer casts.

The following additional new options are available in the compilers:

-Ono_inline Disable inlining. This option replaces -zpdebug_inlines.

-memaccess Specifies the memory attributes of the target system.

-nostrict Enables minor extensions to the C and C++ standards.

The changes to the qualifiers to the -apcs option are listed in Table 2-1.

Table 2-1 Procedure call standard qualifiers

ADS form	SDT 2.50/2.51 equivalent
[no]interwork	[no]<u>inter</u>work
[no]ropi	Not available
[no]rwpi	Not available
[no]swstackcheck	[no]<u>swst</u>ackcheck
Obsolete. Now always nofp.	[no]fp
No direct equivalent. For default behavior use -fpu softVFP. For compatibility with legacy SDT objects or libraries, use -fpu softFPA.	softfp
No direct equivalent, use -fpu FPA.	hardfp

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Table 2-1 Procedure call standard qualifiers (continued)

ADS form	SDT 2.50/2.51 equivalent
Not available.	`[no]fpregargs`
Obsolete. Now always `narrow`.	`narrow`, `wide`
No direct equivalent, use `-rwpi`.	`[non]reentrant`

Impact

Update your projects or makefiles to compile with the appropriate options. In most cases you do not have to change your source code to use the new options.

Check the assembler, compiler, and linker options for your new or migrated projects as the defaults for ADS 1.0 are different from the defaults for SDT 2.50/2.51.

Obsolete compiler pragmas

The following pragmas from the ARM Software Development Toolkit are not supported in the compiler:

```
check_memory_accesses
optimize_cross_jump
optimize_cse
optimize_multiple_loads
optimise_scheduling
side_effects
continue_after_hash_error
debug_inlines
force_toplevel
include_only_once
```

Impact

If you are creating new applications, there is no impact. If you are recompiling existing applications, ensure that the appropriate build options are specified to the compiler. Remove any obsolete pragmas from your source code and replace them, where necessary, with equivalent compiler options.

Obsolete compiler options

The following options from the ARM Software Development Toolkit are not supported in the compiler:

`-zp`*`name`*

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Select pragma from command line.

`-zi`*`number`*

Replaced by `-Ospace` and `-Otime`.

`-gx`*`letter`*

Replaced by the `-O[0|1|2]` options.

`-dwarf`	Use `-dwarf2` (or `-dwarf1`).
`-aof`	Output AOF.
`-asd`	Output ASD format debug tables.
`-MD`	Generate APM dependency.
`-cfront`	Select Cfront-style C++.
`-pcc`	Select Berkeley PCC.
`-fussy`	Synonym for `-strict`.
`-pedantic`	Synonym for `-strict`.
`-fw`	Make string literals writable.

`-za`*`number`*

Use `-memaccess` instead. The default behavior for ADS 1.0 is for `LDR` to access only word-aligned addresses (`-za1`).

`-zt` Fault tentative declarations. This is now the default unless `-strict` is specified.

`-zz`*`number`*

Default is `-zzt0`.

`-zzt`*`number`*

Combines the `-zt` and `-zz` options.

`-zap` Specify whether pointers to structures are assumed to be aligned on at least the minimum byte alignment boundaries set by `-zas`. The behavior for ADS 1.0 is `-zap0`.

`-zat` Default is `-zat1`.

`-zr`*`number`*

Set the number of register values transferred by `LDM` and `STM` instructions. The compilers never generate `LDM` or `STM` instructions that transfer more than nine register values for either ARM code or Thumb code.

`-fz` This is now the default.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Impact

If you are creating new applications, there is no impact. If you are recompiling existing applications, ensure that the appropriate build options are specified to the compiler. Remove any obsolete options from your make files and replace them, where necessary, with equivalent options. Check the assembler, compiler, and linker options for your new or migrated projects as the defaults for ADS 1.0 are different from the defaults for the SDT 2.50/2.51.

Deprecated compiler options

The following options are deprecated and will not be supported in future versions of the compiler:

`-dwarf1` Use `-dwarf2`.

`-proc, -arch` Select processor or architecture. Use `-cpu` instead.

`-zas`*`num`* Align structures on at least a *`num`*-byte boundary (1, 2, 4, or 8). The default is now 1 (align only as strictly as the contents of the structure require).

Impact

You can still output DWARF1 debug tables. However, the functionality of these output files when used with the new debuggers might be reduced. Use DWARF2 format for new projects and update your existing tools to use the DWARF2 format.

Obsolete ARM-specific language extensions

The following language extensions are obsolete:

`__global_freg`

This language extension is not required.

`___weak (three underscores)`

This was a synonym for `__weak` (two underscores) in SDT 2.50/2.51. Use `__weak`.

`__softfp` This is a storage class specifier you can use in the declaration of a function to indicate that the function has a software floating-point interface (a **`double`** parameter passed in two integer registers, a **`double`** result returned in a0, a1) even though its implementation may use floating-point instructions. Use this to create ARM-state, VFP-using (or FPA-using) functions that you can call directly from Thumb state (in which floating-point instructions are inaccessible).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Obsolete and new predefined macros

The obsolete predefined macros are listed in Table 2-2.

Table 2-2 Obsolete predefined macros

Predefine	Status	Comments
`__CLK_TCK`	Obsolete	C library use only.
`__APCS_32`	Obsolete	Relates to obsolete APCS/TPCS. No ATPCS equivalent.
`__APCS_FPREGARGS`	Obsolete	Relates to obsolete APCS/TPCS. No ATPCS equivalent.
`__APCS_NOFP`	Obsolete	Relates to obsolete APCS/TPCS. No ATPCS equivalent.
`__APCS_REENT`	Obsolete	Relates to obsolete APCS/TPCS. No ATPCS equivalent.
`__APCS_NOSWST`	Obsolete	Relates to obsolete APCS/TPCS. Use new __APCS_SWST.
`__CFRONT_LIKE`	Obsolete	The option `-cfront` is now obsolete.
`__DIALECT_PCC`	Obsolete	The option `-pcc` is now obsolete.
`__DIALECT_FUSSY`	Obsolete	The option `-fussy` is now obsolete.

The new predefined macros are listed in Table 2-3.

Table 2-3 New predefined macros

Predefine	Status	Comments
`__CC_ARM`	New	Always defined.
`__STRICT_ANSI__`	New	Set by `-strict`.
`__embedded_cplusplus`	New	Set by `-embeddedcplusplus`.
`__APCS_ROPI`	New	Set by `-apcs /ropi`.
`__APCS_RWPI`	New	Set by `-apcs /rwpi`.
`__APCS_SWST`	New	Set by `-apcs /swst`.
`__FEATURE_SIGNED_CHAR`	New	Set by `-zc`.
`__OPTIMISE_SPACE`	New	Set by `-Ospace`.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Table 2-3 New predefined macros (continued)

Predefine	Status	Comments
`__OPTIMISE_TIME`	New	Set by `-Otime`.
`__TARGET_FPU`	New	Target Floating Point Unit
`__TARGET_FEATURE_DSPMUL`	New	Set if ARM9E multiplier available.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

2.3.4 Changed assembler behavior

This section describes assembler behavior that is changed, deprecated, or obsolete. Obsolete features are identified explicitly. Their use is faulted in ADS. Deprecated features will be made obsolete in future releases. Their use is warned about in ADS.

New or changed assembler behavior

The following enhancements and changes are available in the assembler:

- The assembler provides new ATPCS command-line options similar to those for the compilers.
- The default floating-point option is `-fpu softvfp`.
- A new default software stack checking option of `-swstna` is introduced for code that is compatible with both software stack checking code and non software stack checking code. This option makes explicit the behavior of the assembler. There is no change to the default behavior.
- The assembler always outputs ELF object code. AOF is no longer supported.
- The assembler requires the dollar ($) and double quotation (") characters to be doubled when they are included in string literals. SDT 2.50/2.51 required only a single dollar or double quote character. For example, the following statement in SDT:

  ```
  copyloadsym SETS  "|Load$$":CC:namecp:CC:"$$Base|"
  ```

  must in ADS be:

  ```
  copyloadsym SETS  "|Load$$$$":CC:namecp:CC:"$$$$Base|"
  ```

- The new `-memaccess` option specifies the memory attributes of the target system.
- The `-list` option now accepts an argument of - to select stdout.
- DWARF2 stack-unwinding descriptions can be, and are recommended to be, produced by the use of new directives.
- The assembler supports the new ARM9E and ARM10 instructions. Use one of ARM9E, ARM10TDMI™, ARM1020T, or ARM10200™ with the `-cpu` option.
- Support is provided for VFP in both scalar and vector mode.
- New directives `DCQ` and `DCQU` define a 64-bit integer value. `DCQ` is aligned to a 32-bit boundary while `DCQU` is unaligned (byte boundary).
- The `DCFD`, `DCFDU`, `DCFS` and `DCFSU` directives now also accept a hex-constant form of operand that specifies the IEEE bit-pattern of the value.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

- There are new synonyms `FIELD` and `SPACE` for `#` and `%` directives.
- Directives are now accepted in all upper case, or all lower case, but not a mixture. Previously, only the upper case form was accepted.
- The `EXPORT` directive may have a new attribute, `WEAK`. This defines the exported symbol as a `WEAK` symbol in ELF.
- The semantics of the `EXTERN` and `IMPORT` directives have changed and they are no longer synonyms. An unused `IMPORT` generates an undefined global symbol, whereas an unused `EXTERN` generates no symbol. (In SDT 2.50/2.51 an unused `EXTERN` or `IMPORT` symbol was made `WEAK`).
- The `AREA` directive has a new attribute, `ASSOC=` *`area_name`*) that requires this `AREA` to be included in a link step whenever the associate area (named by the `ASSOC=`*`area_name`*) is included. The assembler implements the requirement by generating an `R_ARM_NONE` relocation at offset 0 of area *`area_name`*, relative to the section symbol for the area defined by the `AREA` directive.
- The new directive `REQUIRE`*`area_name`* requires *`area_name`*. to be included in any link step that includes the requiring section. The assembler implements the requirement by generating an `R_ARM_NONE` relocation in the current section to the required *`area_name`*.
- The `DCD` directive now accepts expressions evaluating the difference between a label in another section and a position in the current section.
- The `DCW` and `DCB` directives now accept expressions including an external symbol.
- The new `DCDO` directive treats label operands as sb-relative.
- The literal-using, pseudo-instruction forms of load and store instructions (for example, `LDR rx,=yyy`) can now take external symbols as immediate values (`yyy`).
- The ARM instructions of the form *`data-processing-op`* `rd,rn,#sym` can now take external symbols as immediate operands.
- ARM and Thumb SWI instructions can now take external symbols as immediate operands.
- If you select a cpu or architecture that does not support Thumb, an attempt to generate Thumb code will generate an error message. For example `armasm -cpu 4` will not accept Thumb instructions but `armasm -cpu 4T` will.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Features of the SDT assembler not supported

The following assembly language features are no longer supported and are faulted:

- `AREA` directive with attribute `ABS`, `BASED`, `A32bit`, `HALFWORD`, `INTERWORK`, `PIC`, `REENTRANT`
  - `ABS` has been withdrawn because it conflicts with the linker scatter-loading mechanism. An `AREA` previously declared `ABS` should now be placed using a scatter-loading description
  - `BASED` has been withdrawn because it was needed only for the old shared library mechanism that is now obsolete. No workaround is necessary.
  - `A32bit` has been withdrawn as it was needed only to distinguish 32 bit mode code from 26 bit mode code and 26 bit mode is now obsolete.
  - `INTERWORK` and `PIC` have been withdrawn as the ATPCS and architecture are now always specified on the command line. Any occurrences of these attributes should be deleted, and replaced by the corresponding new `-apcs` command line qualifiers.
- value 32 as operand to the `ALIGN` area attribute. The assembler accepted 32 as operand to `ALIGN` even though it was not useful. The only address that satisfies `ALIGN=32` is 0, and if that is the desired behavior it can be expressed by using a scatter-loading description to place the `AREA` at address 0
- `IMPORT` directive with attribute `FPREGARGS`. The `FPREGARGS` attribute had no effect and has been removed.
- `EXPORT` directive with attribute `FPREGARGS`, and `LEAF`.
  - The `FPREGARGS` attribute had no effect. The workaround is to delete it from assembly source.
  - `The LEAF` attribute was needed only for the old shared library mechanism that is now obsolete. The workaround is to remove it.
- `ADR` pseudo-instructions with out-of-area symbol operands. The workaround is to load out-of-area addresses into registers using `LDR`.

Deprecated assembler options

The following options are deprecated and will not be supported in future versions of the assembler:

`-dwarf1` DWARF1 debug tables will not be supported in future versions.

`-proc` Select processor (use `-cpu` instead).

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

`-arch` Select architecture (use `-cpu` instead).

Impact

Use DWARF2 format for new projects and update your existing tools to use the DWARF2 format.

2.3.5 Changed linker behavior

This section describes linker behavior that is changed, deprecated, or obsolete. Obsolete features are identified explicitly. Their use is faulted in ADS. Deprecated features will be made obsolete in future releases. Their use is warned about in ADS.

New or changed linker behavior

The following new or significantly changed options are available in the linker:

- The linker is now an ELF-only linker.
- The syntax of the `-remove` command has been expanded to include section attribute qualifiers. This is backwardly compatible with SDT 2.50/2.51.

  The linker now has `-remove` as its default option. The SDT 2.50/2.51 default was `-noremove`. The `-remove` option is strongly recommended with C++ in order to reduce code size. Use the new linker option `-keep` if you want to keep sections that are not referenced.
- The syntax of `-first` and `-last` has been changed to identify both object and section name, not just section name as in SDT 2.50/2.51. There is no backward compatibility with SDT 2.50/2.51.
- The syntax of the `-entry` command has been changed to allow more flexible selection. Only one entry point can be specified to the linker. There is some backward compatibility with SDT 2.50/2.51.
- The `veneers` argument has been added to the `-info` option.
- The linker now generates conventionally named region-related symbols for non scatter-loaded images, in a similar way to those generated for scatter-loaded images.

The following new armlink options have been added:

`-partial`	Generate a partially-linked ELF object
`-ropi`	RO execution region is position-independent
`-rwpi`	RW execution region is position-independent
`-split`	Image has two load regions

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

`-keep`	Specify sections to be retained even if unused
`-locals`	Add local symbols to image symbol table
`-nolocals`	Remove local symbols from image symbol table
`-xreffrom`	List section cross references in image from a section
`-xrefto`	List section cross references in image to a section
`-strict`	Strict compliance to build attribute rules
`-symdefs`	Create, or read, a list of symbol definitions.

Obsolete linker options

The following options from the ARM Software Development Toolkit are not supported in the linker:

`-aof`	Create output in AOF format
`-aif`	Create output in AIF format
`-aif -bin`	Create output in AIF BIN format
`-bin`	Create output in BIN format
`-base`	Alias for ro-base
`-data`	Alias for rw-base
`-dupok`	Allow multiple definitions
`-[no]case`	Case sensitive/insensitive matching
`-match`	Symbol matching options
`-nozeropad`	Do not include ZI section in binary images
`-info interwork`	Output information on interworking
`-u`	Match all unresolved symbols.

Impact

If you are creating new applications, there is no impact. If you are relinking existing applications and libraries, ensure that the desired build options are specified to the assembler, compiler and linker. Remove any obsolete options from your make files and replace them, where necessary, with equivalent options. Check the assembler, compiler, and linker options for your new or migrated projects as the defaults for ADS 1.0 are different from the defaults for the SDT 2.50/2.51.

2.3.6 Obsolete components and standards

This section describes components of SDT 2.50/2.51 that are not available in ADS.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

APM

APM is not provided.

Impact

Use the CodeWarrior IDE or a make utility.

Armmake

Armmake is not provided. There is no longer a need to rebuild the C Libraries, therefore the ARM-specific make utility has been removed.

Impact

None. Use nmake, make, or gnumake if you want to use a make utility.

Armlib

The ARM librarian, armlib is not provided. It has been replace by a new utility, armar, that creates ELF `ar` files. armar provides similar functionality to armlib, but supports ELF instead of AOF.

Decaof and Decaxf

Decaof and decaxf are not provided.

Impact

The fromELF utility provides equivalent functionality for ELF formats

DWARF1

The compilation tools produce DWARF2 debug table formats by default. The compiler and assembler can still produce DWARF1 for compatibility with third party tools that require DWARF1, although DWARF1 will only support debugging for C compiler images produced with the `-O0` option and will not support debugging of C++ images.

DWARF1 is deprecated and will be removed in a future release of ADS

Impact

Use DWARF2 format.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

26-bit addressing

ADS does not support 26-bit addressing. Removal of 26-bit support has enabled a more efficient ATPCS to be designed.

Impact

Continue to use SDT2.50/2.51 if you need 26-bit support.

AOF, AIF, IHF, and Plain Binary image formats

The SDT 2.50/2.51 linker gave warnings when asked to generate an AIF image, a binary AIF image, an IHF image, or a plain binary image. The ADS linker refuses to generate these images and is now a pure ELF linker. Although the linker is capable of processing AOF files, you are strongly recommended not to link with old AOF files because of changes to both the Procedure Call Standard and changes to debug tables.

Impact

Use the fromelf tool to translate the ELF image into non-ELF formats such as AIF, Plain binary, Extended Intellec Hex (IHF), Motorola 32 bit S-record, Intel Hex 32.

Future releases of the linker will not allow AOF input files.

RDI 1.50

A new variant of the Remote Debug Interface (RDI 1.5.1) is introduced in ADS. The version used in SDT 2.50/2.51 was 1.5. See *Debuggers* on page 2-26 for details of RDI 1.51.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Chapter 3
Creating an Application

This chapter describes how to create an application using ADS. It contains the following sections:

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.1 Using the CodeWarrior IDE

The CodeWarrior IDE provides a simple, versatile, graphical user interface for managing your software development projects. You can use CodeWarrior for the ARM Developer Suite to develop C, C++, and ARM assembly language code targeted at ARM processors. The CodeWarrior IDE enables you to configure the ARM tools to compile, assemble, and link your project code.

The CodeWarrior IDE enables you to organize source code files, library files, other files, and configuration settings into a *project*. Each project enables you to create and manage multiple *build targets*. A build target is the collection of build settings and files that determines the output that is created when you build your project. Build targets can share files in the same project, while using their own build settings.

Note

A build target is distinct from a *target system*, such as an ARM development board. For example, you can compile a debugging build target and an optimized build target of code targeted at hardware based on an ARM7TDMI.

CodeWarrior for the ARM Developer Suite provides preconfigured *project stationery* files for common project types, including:

- ARM Executable Image
- ARM Object Library
- Thumb Executable Image
- Thumb Object Library
- Thumb/ARM Interworking Image.

You can use the project stationery as a template when you create your own projects.

The non-interworking ARM project stationery files define three build targets. The interworking project stationery defines an additional three build targets to compile Thumb-targeted code. The basic build targets for each of the stationery projects are:

Debug This build target is configured to build output binaries that are fully debuggable, at the expense of optimization.

Release This build target is configured to build output binaries that are fully optimized, at the expense of debug information.

DebugRel This build target is configured to build output binaries that provide adequate optimization, and give a good debug view.

For more information on using the CodeWarrior IDE to create complex, dependent build target relationships, see the *CodeWarrior IDE Guide*.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.1.1 Creating and building a simple project

This section describes how to create and build a simple project. It uses source files from the `dhryansi` example supplied with ADS 1.1 to give an introduction to configuring tool options and using build targets in the CodeWarrior IDE.

——— **Note** ———

This example assumes that you have installed the example code supplied with ADS 1.1, and that you have installed in the default installation directory. Example code is installed by default unless you have chosen a minimal install or a custom install.

————————

The following sections give a summary of how to:

- Create a new project using ARM project stationery
- Add source files to your project
- Configure the build target settings for your project
- Compile and link an executable image.
- Execute the AXD debugger to debug your image.

Creating a new project from ARM project stationery

To create a new project, and compile and link an application using the CodeWarrior IDE:

1. Select **Programs** → **ARM Developer Suite** → **CodeWarrior for ARM Developer Suite v1.1** from the Windows **Start** menu to start the CodeWarrior IDE.
2. Select **New…** from the **File** menu. A New dialog is displayed (see Figure 3-1 on page 3-4).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

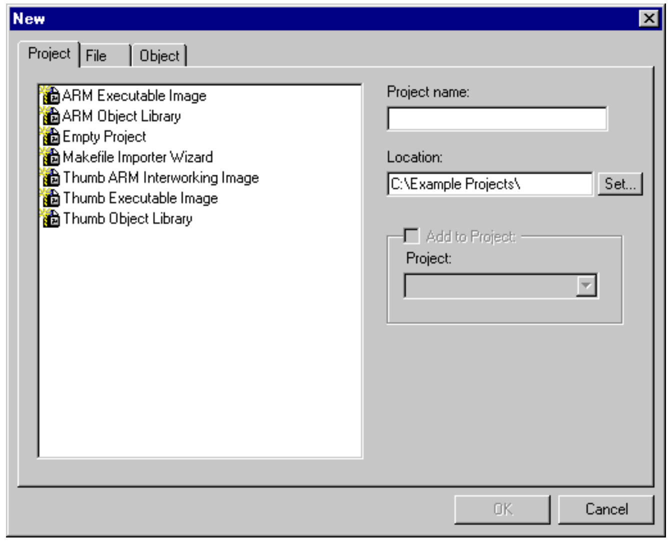

Figure 3-1 New dialog

3. Ensure that the **Project** tab is selected. The available ARM project stationery is listed in the left of the dialog (see Figure 3-1), together with the Empty Project stationery and the Makefile Importer Wizard.

   See the *CodeWarrior IDE Guide* for more information on using empty projects and the Makefile Importer Wizard.

4. Select **ARM Executable Image** from the list of project stationery.

5. Click the **Set…** button next to the Location field. A Create New Project dialog is displayed (Figure 3-2).

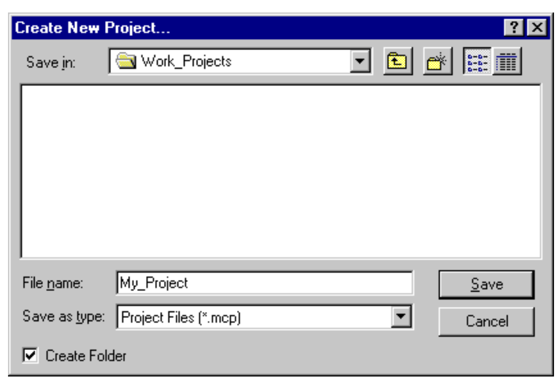

Figure 3-2 Create New Project dialog

Copyright © 1999, 2000 ARM Limited. All rights reserved.

6. Navigate to the directory where you want to save the project and enter a project name, for example `My_Project`. Leave the **Create Folder** checkbox selected.

7. Click **Save**. The CodeWarrior IDE sets the Project Name field and Location path in the New dialog box. The Location path is used as a default when you create additional projects.

8. Click **OK**. The CodeWarrior IDE creates a new project based on the ARM Executable Image project stationery, and displays a new project window with the Files view selected (Figure 3-3).

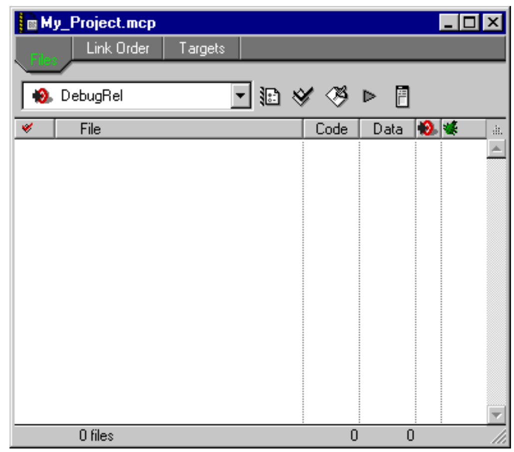

Figure 3-3 New project

Adding source files to the project

Projects created from ARM project stationery do not contain source files. This section describes how to add the source files from the `dhryansi` example.

To add source files to a project:

1. Ensure that the project window is the active window.

2. Select **Add Files…** from the **Project** menu. A Select files to add… dialog is displayed.

3. Navigate to the `dhryansi` directory in the *`install_directory`*`\Examples` directory and Shift-click on `dhry_1.c` and `dhry_2.c` to select them (Figure 3-4).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

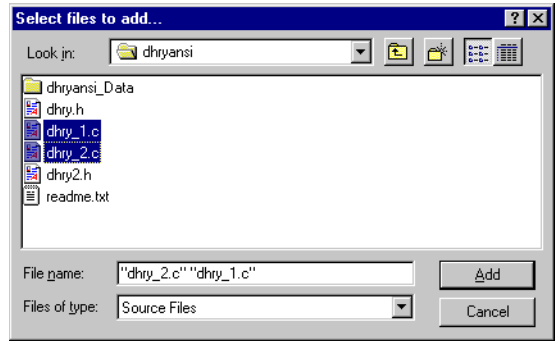

Figure 3-4 Select files to add… dialog

4. Click **Add**. The CodeWarrior IDE displays an Add Files dialog (Figure 3-5). The dialog contains a checkbox for each build target defined in the current project. In this example, the dialog contains three checkboxes corresponding to the three build targets defined in the ARM Executable Image project stationery.

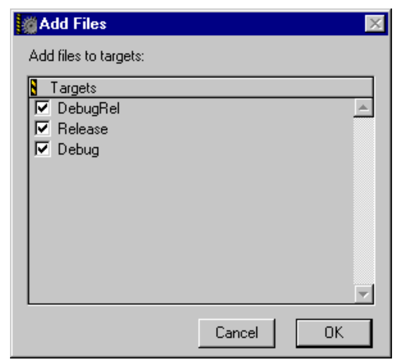

Figure 3-5 Add Files

5. Leave all the build target checkboxes selected and click **OK**. The CodeWarrior IDE adds the source files to each target in the project and displays a Project Messages window to inform you that the directory containing the source files has been added to the access paths for each build target (Figure 3-6).

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

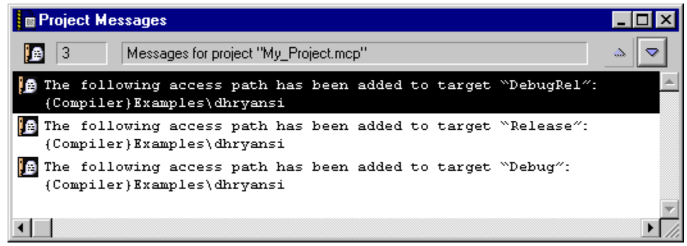

Figure 3-6 Project messages window

The access paths for each build target define the directories that will be searched for source and header files. See the *CodeWarrior IDE Guide* for details.

Note

You do not need to explicitly add the header files for the `dhryansi` project because the CodeWarrior IDE locates them in the newly added access path. However, you can add header files explicitly if you want.

6. Ensure that the **Files** tab is selected in the project window. The project window displays all the source files in the project. (Figure 3-7). See the *CodeWarrior IDE Guide* for more information on what is displayed when you click the **Link Order** tab and the **Targets** tab.

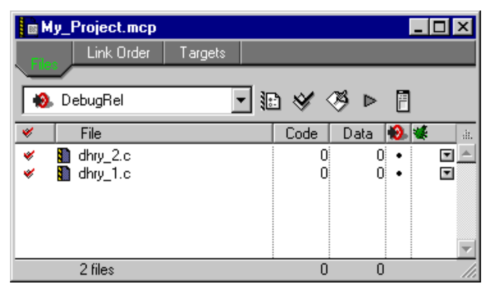

Figure 3-7 Source files in Files view

Configuring the project build targets

This section describes how to configure your example project so that the example `dhryansi` files compile, and the project build settings are the same as those in the supplied `dhryansi` example project. It describes one way of selecting build targets, and shows how different build target settings can be used in the same project. See the *CodeWarrior IDE Guide* for a complete description of build targets.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Build target settings must be selected separately for each build target in your project. To set build target options for the `dhryansi` example:

1. Ensure that the DebugRel build target is currently selected. By default, the DebugRel build target is selected when you create a new project based on the ARM project stationery. The currently selected build target is displayed in the **Build Target** pop-up menu in the project toolbar (Figure 3-8 on page 3-8).

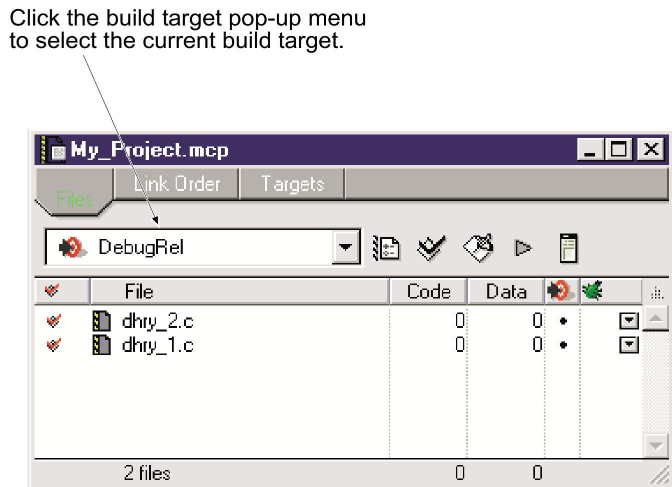

Figure 3-8 Currently selected build target

2. Select **DebugRel Settings…** from the **Edit** menu. The name of this menu item changes depending on the name of the currently selected build target. The CodeWarrior IDE displays the DebugRel Target Settings panel (Figure 3-9 on page 3-9). All the target-specific settings are accessible through configuration panels listed at the left of the panel.

   Note

   Many configuration options are optional, however you should review the target settings for each build target in your project to ensure that they are appropriate for your target hardware, and your development requirements. See the chapter on configuring a build target in the *CodeWarrior IDE Guide* for configuration recommendations.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

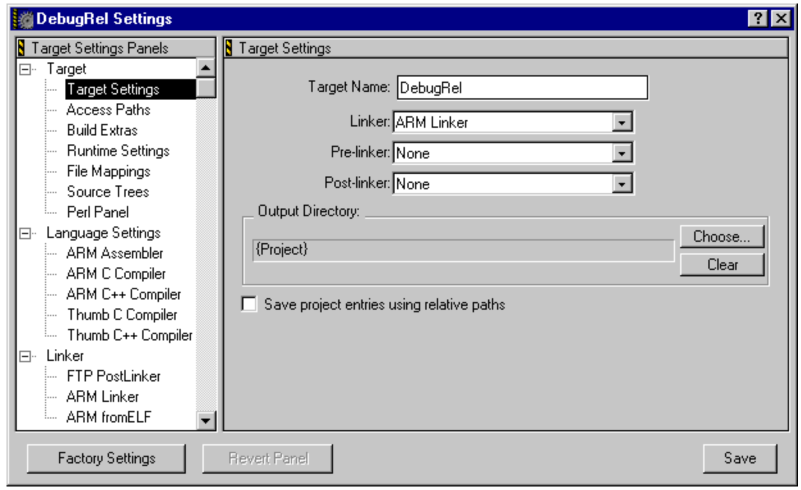

Figure 3-9 DebugRel Settings

3. Click the ARM C Compiler entry in the Target Settings Panels list to display the configuration panel for the C compilers. The Target and Source panel is displayed (Figure 3-10 on page 3-9). The panel consists of a number of tabbed panes containing groups of configuration options. For this example, the `dhryansi` source requires a predefined macro be set before it will compile.

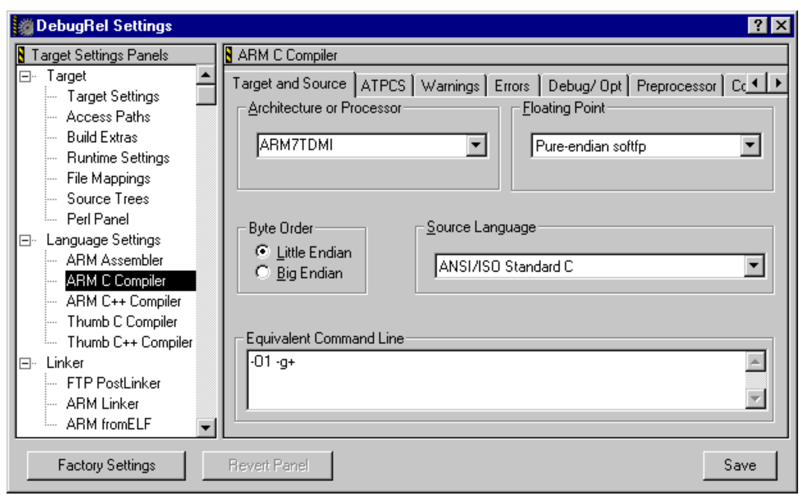

Figure 3-10 ARM C compiler panel

4. Click the Preprocessor tab to display a list of predefined macros (Figure 3-11).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

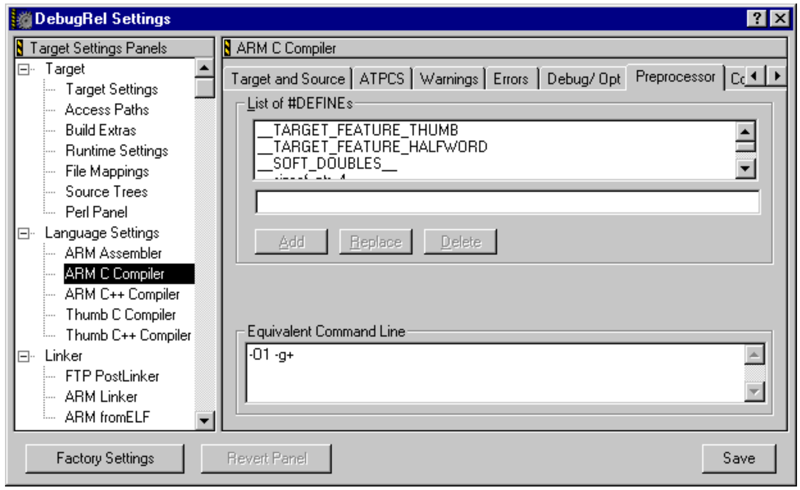

Figure 3-11 ARM C compiler preprocessor pane

5. Type MSC_CLOCK into the text field beneath the List of #DEFINES and click **Add** to define the MSC_CLOCK macro. The CodeWarrior IDE adds MSC_CLOCK to the List of #DEFINES. The Equivalent Command Line text box displays the compiler command-line option required to define MSC_CLOCK (Figure 3-12).

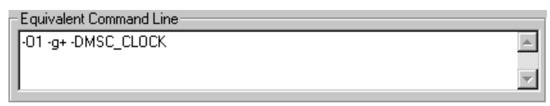

Figure 3-12 MSC_CLOCK defined

6. Click **Save** to save your changes, and close the DebugRel Settings panel.

At this point you have defined the MSC_CLOCK macro for the DebugRel build target only. You must also define the MSC_CLOCK macro for the Release and Debug build targets if you want to use them. To select the Release build target:

1. Ensure that the Project window is currently active.

2. Click the Current Target pop-up menu to display the list of defined build targets (see Figure 3-8 on page 3-8).

3. Select Release from the list of build targets to change the current build target.

4. Apply the steps you followed above to define MSC_CLOCK the Release build target.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

——— **Note** ———

You can also cut and paste build target settings into the Equivalent Command Line text box. Press the Enter key to set the options and update the panel controls. Be careful not to copy command-line options that are inappropriate, such as the optimization and debug settings, from one build target to another.

———————

Leave the Release Target settings panel open after you have saved your changes.

5. Click on the **Debug/Opt** tab to display Debug and Optimization options for the Release build target (Figure 3-13).

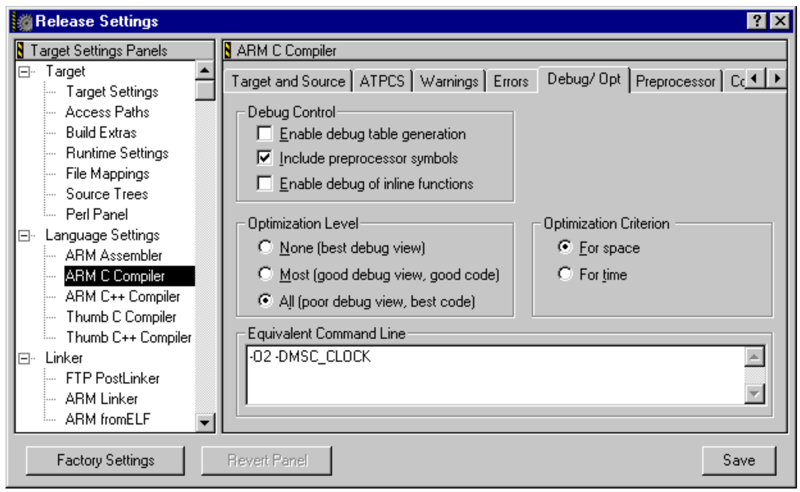

Figure 3-13 Debug/Opt configuration panel

6. Select the **For time** Optimization Criterion button. The Equivalent Command Line text box reflects the change.

7. Click **Save** to save your settings.

8. Define `MSC_CLOCK` in the Debug build target in the same way as you have for the DebugRel and Release build targets.

   Your project is now equivalent to the `dhryansi` example project supplied with the ARM Developer Suite.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Note

This example has shown how to use the configuration dialogs to set options for individual build targets. There are configuration panels available for most of the ADS toolchain, including the linker, fromELF, and the assembler. You can use the configuration panels to specify most options available in the tools, including:

- procedure call options
- the structure of output images
- the linker and postlinker to use
- the ARM debugger to call from the CodeWarrior IDE.

See the chapter on configuring a build target in the *CodeWarrior IDE Guide* for a complete description of build target options.

Building the project

The **Project** menu contains a number of commands to compile, or compile and link your project files. These commands apply only to the current build target. To compile and link the example project:

1. Ensure that the project window is the currently active window.
2. Select the build target you want to build (see Figure 3-8 on page 3-8). For this example, select the DebugRel build target.
3. Select **Make** from the **Project** menu. The CodeWarrior IDE builds the project by:
    - compiling newly added, modified, and touched source files to produce ELF object files
    - linking object files and libraries to produce an ELF image file, or a partially linked object
    - performing any postlink operations that you have defined for your build target, such as calling fromELF to convert an ELF image file to another format.

      Note

      In the `dhryansi` example there is no postlink operation.

    If the project has already been compiled using a command such as **Bring Up To Date** or **Compile**, the **Make** command performs only the link and postlink steps.

    The compiler displays build information, errors, and warnings for the build in a messages window.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Debugging the project

By default, the ARM project stationery is configured to call the AXD debugger to debug and run images built from the CodeWarrior IDE. You can configure the debugger to be called using the ARM Debugger configuration panels for each build target. See the *CodeWarrior IDE Guide* for details.

To execute and debug your example project:

1. Ensure that the project window is the currently active window.
2. Select the build target you want to debug. The **Debug** command applies only to the current build target.
3. Select **Debug** from the **Project** menu. The CodeWarrior IDE compiles and links any source files that are not up to date, and calls the AXD debugger to load and execute the image. See *Debugging the application with AXD* on page 3-26 for more information on using AXD.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.2 Building from the command line

This section describes how to build an application from the command line. From the command line, you can access:

- the compilers
- the assembler
- the linker
- the CodeWarrior IDE.

3.2.1 Using the compilers from the command line

There are four compiler variants as shown in Table 3-1:

Table 3-1 Compiler variants

Compiler name	Compiler variant	Source language	Compiler output
armcc	C	C	32-bit ARM code
tcc	C	C	16-bit Thumb code
armcpp	C++	C or C++	32-bit ARM code
tcpp	C++	C or C++	16-bit Thumb code

Compiler syntax

The command for invoking the ARM compilers is:

```
compiler [PCS-options] [source-language] [search-paths] [preprocessor-options]
[output-format] [target-options] [debug-options] [code-generation-options]
[warning-options] [additional-checks] [error-options] [source]
```

Refer to the *ADS Compiler, Linker, and Utilities Guide* and the *ADS Developer Guide* for more information.

Building an example

Sample C source code for a simple application is in `install_directory\Examples\rom\embed\main.c`.

To build the example from the command line:

1. Compile the C file `main.c` with the following command with either:

Copyright © 1999, 2000 ARM Limited. All rights reserved.

`armcc -g -O1 -c main.c` (for ARM)

`tcc -g -O1 -c main.c` (for Thumb)

where:

`-g` Tells the compiler to add debug tables.

`-O1` Tells the compiler select good optimization.

`-c` Tells the compiler to compile only (not to link).

2. Link the image using the following command:

   `armlink main.o -o embed.axf`

   where:

   `-o` Specifies the output file as `embed.axf`.

3. Use ARMulator to test the image or download the image to a development board and use Multi-ICE.

For more information on using assembly language, C, C++ and the linker to create applications, see the *ADS Developer Guide* and *ADS Compiler, Linker, and Utilities Guide*.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.2.2 Using the CodeWarrior IDE from the command line

In some cases you might not require the Graphical User Interface of the CodeWarrior IDE, for example, when a project is part of a larger system that must be built automatically without user interaction.

`CMDIDE.EXE` is a console window program that can be started from the command line to build project files that have been created and edited with the CodeWarrior IDE. `CMDIDE.EXE` invokes the CodeWarrior IDE, passes the proper parameters to produce a build, and waits for the IDE to finish its operation.

The command-line arguments are:

Projectname
: Specifies the project to use.

/t*Targetname*
: Specifies a target to become the current target.

/r
: Removes the objects of the current target before building.

/b
: Builds the current target.

/c
: Closes the project after building.

/q
: Quits the IDE after building.

/v[y|n|a]
: Option for converting projects on open:
  - y — Convert without asking.
  - n — Do not convert.
  - a — Ask whether to convert.

/s
: Forces the command line to be processed in a new instance of the IDE (rather than using a current instance).

If more than one project document is specified to be opened in the command line, the `/t` target and `/b` build command flags apply to the first project document found in the list of documents. If no project is specified in the command line, it uses the usual logic in the IDE (front project or default project) to select the project to build.

For example, to build the `dhryansi` project, change directory to the `dhryansi` example directory and type:

```
cmdide dhryansi.mcp /t DebugRel /c /b
```

If no target is specified `cmdide` uses whatever the current target is for that project.

All build commands are executed in a different process from the one launched from the command line. The original process returns when the command line has been completely processed and the build has completed.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

The following codes are returned and can be tested using the `IF ERRORLEVEL` instruction in a batch file:

`0`	No error
`1`	Error opening file
`2`	Project is not open
`3`	IDE is already building
`4`	Invalid target name (for `/t` flag)
`5`	Error changing current target
`6`	Error removing objects
`7`	Build was canceled
`8`	Build failed
`9`	Process aborted.

Note

Though `IDE.EXE` understands the same parameters as `CMDIDE.EXE`, it is particularly important on Windows 95 or Windows 98 to use `CMDIDE.EXE` to ensure that builds are correctly serialized rather than executed all at once.

3.2.3 Debugging from the command line

You can use the ARM symbolic debugger (armsd) to debug applications from the command line.

AXD can also be driven from the command line. This is useful for batch testing, for example.

See the *ADS Debuggers Guide* for more information on using the debuggers.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.2.4 Using the assembler from the command line

The basic syntax to use the ARM assembler (armasm) from the command-line is:

`armasm -list` *`listingfile inputfile`*

For example, to assemble the code in a file called `myfile.s`, type:

```
armasm -list myfile.lst myfile.s
```

This produces an object file called `myfile.o`, and a listing file called `myfile.lst`.

For full details of the command-line options and syntax, refer to the Assembler chapter in *ADS Compiler, Linker, and Utilities Guide*.

Example 3-1 shows a small interworking ARM/Thumb assembly language program. You can use it to explore the use of the assembler, linker, and the ARM symbolic debugger.

Example 3-1

```
     AREA     AddReg,CODE,READONLY         ; Name this block of code.
     ENTRY                  ; Mark first instruction to call.
main
     ADR r0, ThumbProg + 1    ; Generate branch target address and set bit 0
                            ; hence arrive at target in Thumb state.
     BX r0                  ; Branch and exchange to ThumbProg.
     CODE16                 ; Subsequent instructions are Thumb code.
ThumbProg
     MOV r2, #2             ; Load r2 with value 2.
     MOV r3, #3             ; Load r3 with value 3.
     ADD r2, r2, r3         ; r2 = r2 + r3
     ADR r0, ARMProg        ; Generate branch target address with bit 0 zero.
     BX r0                  ; Branch and exchange to ARMProg.
     CODE32                 ; Subsequent instructions are ARM code.
ARMProg
     MOV r4, #4
     MOV r5, #5
     ADD r4, r4, r5
stop MOV r0, #0x18          ; angel_SWIreason_ReportException
     LDR r1, =0x20026       ; ADP_Stopped_ApplicationExit
     SWI 0x0123456          ; ARM semihosting SWI

     END                    ; Mark end of this file.
```


Copyright © 1999, 2000 ARM Limited. All rights reserved.

Building the example

To build the example:

1. Enter the code using any text editor and save the file in your current working directory as `addreg.s`.
2. Type `armasm -list addreg.lst addreg.s` at the command prompt to assemble the source file.
3. Type `armlink addreg.o -o addreg` to link the file.

Running the example in the debugger

To load and run the example in the debugger:

1. Type `armsd addreg` to load the module into the command-line debugger.
2. Type `step` to step through the rest of the program one instruction at a time. After each instruction, you can type `reg` to display the registers.

When the program terminates, to return to the command line, type `quit`.

This example contains both ARM and Thumb code. As you step through the program you can see the T-bit in the *Current Program Status Register* (CPSR) changing between a lowercase `t` and an uppercase T. This indicates the change between ARM and Thumb state.

For further details on ARM and Thumb assembly language programing, see the *ADS Assembler Guide*.

3.2.5 Setting linker options from the command line

The ARM linker, armlink, enables you to:

- link a collection of objects and libraries into an executable ELF image
- partially link a collection of objects into an object that can be used as input for a future link step
- specify where the code and data will be located in memory
- produce debug and reference information about the linked files.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Objects consist of input sections that contain code, initialized data, or the locations of memory that must be set to zero. Input sections can be *read-only* (RO), *read/write* (RW), or *zero-initialized* (ZI) These attributes are used by armlink to group input sections into bigger building blocks called output sections, regions and images. Output sections are approximately equivalent to ELF segments.

The default output from the linker is a non-relocatable image where the code starts at `0x8000` and the data section is placed immediately after the code. You can specify exactly where the code and data sections are located by using linker options or a scatter-load description file.

Linker input and output

Input to armlink consists of:

- one or more object files in ELF Object Format
- optionally, one or more libraries created by armar.

Output from a successful invocation of armlink is one of the following:

- an executable image in ELF executable format
- a partially linked object in ELF object format.

For simple images, ELF executable files contain segments that are approximately equivalent to RO and RW output sections in the image. An ELF executable file also has ELF sections that contain the image output sections.

An executable image in ELF executable format can be converted to other file formats by using the fromELF utility.

Linker syntax

The linker command syntax is of the form:

```
armlink [-help_options] [-output_options] [-via_options] [-memory_map_options]
[-image_content_options] [-image_info_options] [-diagnostic_options]
```

See the linker chapter in the *ADS Compiler, Linker, and Utilities Guide* for a detailed list of the linker options.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Using linker options to position sections

The following linker options control how sections are arranged in the final image and whether the code and data can be moved to a new location after the application starts:

`-ropi` This option makes the load and execution region containing the RO output section position-independent. If this option is not used the region is marked as absolute.

`-ro-base` *address*

This option sets the execution addresses of the region containing the RO output section at *address*. The default address is `0x8000`.

`-rw-base` *address*

This option sets the execution addresses of the region containing the RW output section at *address*. The default address is at the end of the RW section.

`-rwpi` This option makes the load and execution region containing the RW and ZI output section position-independent. If this option is not used the region is marked as absolute. The `-rwpi` option is ignored if `-rw-base` is not also used. Usually each writable input section must be read-write position-independent.

`-split` When used with `-ro-base` or `-rw-base`, this option splits the default load region, that contains the RO and RW output sections, into two load regions.

If you want more control over how the sections are placed in an image, use the `-scatter` option and specify a scatter-load description file.

Using scatter-load description files for a simple image

The command-line options (`-ro-base`, `-rw-base`, `-split`, `-ropi`, and `-rwpi`) create simple images.

You can create the more complex images by using the `-scatter` command-line option to specify a scatter-load description file. The `-scatter` option is mutually exclusive with the use of any of the simple memory map options `-ro-base`, `-rw-base`, `-split`, `-ropi`, or `-rwpi`.

For more information on the linker and scatter-load description files, see the *ADS Compiler, Linker, and Utilities Guide* and the Writing Code for ROM chapter in the *ADS Developer Guide*.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.3 Using ARM libraries

The following run-time libraries are provided to support compiled C and C++:

ANSI C The C libraries consist of:

- The functions defined by the ISO C library standard.
- Target-dependent functions used to implement the C library functions in the semihosted execution environment. You can redefine these functions in your own application.
- Helper functions used by the C and C++ compilers.

C++ The C++ libraries contain the functions defined by the ISO C++ library standard. The C++ library depends on the C library for target-specific support and there are no target dependencies in the C++ library. This library consists of:

- the Rogue Wave Standard C++ Library version 2.01.01
- helper functions for the C++ compiler
- additional C++ functions not supported by the Rogue Wave library.

As supplied, the ANSI C libraries use the standard ARM semihosted environment to provide facilities such as file input/output. This environment is supported by the ARMulator, Angel, Multi-ICE, and EmbeddedICE®. You can use the ARM development tools in ADS to develop applications, and then immediately run and debug the applications under the ARMulator or on a development board. See the description of semihosting in the *ADS Debug Target Guide* for more information on the debug environment.

You can re-implement any of the target-dependent functions of the C library as part of your application. This enables you to tailor the C library, and therefore the C++ library, to your own execution environment.

The libraries are installed in two subdirectories within `install_directory\lib`:

`armlib` Contains the variants of the ARM C library, the floating-point arithmetic library, and the math library. The accompanying header files are in `install_directory\include`.

`cpplib` Contains the variants of the Rogue Wave C++ library and supporting C++ functions. The Rogue Wave and supporting C++ functions are collectively referred to as the ARM C++ Libraries. The accompanying header files are installed in `install_directory\include`.

——— **Note** ———

- The ARM C libraries are supplied in binary form only.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

- The ARM libraries should not be modified. If you want to create a new implementation of a library function, place the new function in an object file, or your own library, and include it when you link the application. Your version of the function will be used instead of the standard library version.
- Normally, only a few functions in the ANSI C library require re-implementation in order to create a target-dependent application.
- The source for the Rogue Wave Standard C++ Library is not freely distributable. It can be obtained from Rogue Wave Software Inc., or through ARM Limited, for an additional licence fee. See the Rogue Wave online documentation in `install_directory\Html` for more about the C++ library.

3.3.1 Using the ARM libraries in a semihosted environment

If you are developing an application to run in a semihosted environment for debugging, you must have an execution environment that supports the ARM and Thumb semihosting SWIs and has sufficient memory.

The execution environment can be provided by either:

- using the standard semihosting functionality that is present by default in, for example, ARMulator, Angel, and Multi-ICE
- implementing your own SWI handler for the semihosting SWI.

You do not have to write any new functions or include files if you are using the default semihosting functionality of the library.

3.3.2 Using the ARM libraries in a non-semihosted environment

If you do not want to use any semihosting functionality, you must ensure that either no calls are made to any function that uses semihosting or that such functions are replaced by your own non-semihosted functions.

To build an application that does not use semihosting functionality:

1. Create the source files to implement the target-dependent features.
2. Use `#pragma import use_no_semihosting_swi` to guard the source.
3. Link the new objects with your application.
4. Use the new configuration when creating the target-dependent application.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

You must re-implement functions that the C library uses to insulate itself from target dependencies. For example, if you use `printf()` you must re-implement `fputc()`. If you do not use the higher level input/output functions like `printf()`, you do not have to re-implement the lower level functions like `fputc()`.

If you are building an application for a different execution environment, you can re-implement the target-dependent functions (functions that use the semihosting SWI or that depend on the target memory map). There are no target-dependent functions in the C++ library. See the chapter on libraries in the *ADS Compiler, Linker, and Utilities Guide* for more information.

3.3.3 Building an application without the ARM libraries

Creating an application that has a `main()` function causes the C library initialization functions to be included.

If your application does not have a `main()` function, the C library is not initialized and the following features are not available to your application:

- software stack checking
- low-level `stdio`
- signal-handling functions, `signal()` and `raise()` in `signal.h`
- `atexit()`
- `alloca()`.

You can create an application that consists of customized startup code, instead of the library initialization code, and still use many of the library functions. You must either:

- avoid functions that require initialization
- provide the initialization and low-level support functions.

These applications will not automatically use the full C run-time environment provided by the C library. Even though you are creating an application without the library, some helper functions from the library must be included. There are also many library functions that can be made available with only minor re-implementations. See the chapter on libraries in the *ADS Compiler, Linker, and Utilities Guide* for more information.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.4 Using your own libraries

The ARM librarian, armar, enables sets of ELF object files to be collected together and maintained in libraries. Such a library can then be passed to armlink in place of several object files. However, linking with an object library file does not necessarily produce the same results as linking with all the object files collected into the object library file. This is because armlink processes the input list and libraries differently:

- each object file in the input list appears in the output unconditionally, although unused areas are eliminated if the armlink `-remove` option is specified
- a member of a library file is included in the output only if it is referred to by an object file or a previously processed library file.

To create a new library called `my_lib` and add all the object files in the current directory, type:

```
armar -create my_lib *.o
```

To delete all objects from the library that have a name starting with `sys_`, type:

```
armar -d my_lib sys_*
```

To replace, or add, three objects in the library with the version located in the current directory, type:

```
armar -r my_lib obj1.o obj2.o obj3.o
```

For more information on armar, see the Utilities chapter in the *ADS Compiler, Linker, and Utilities Guide*.

Note

The ARM libraries should not be modified. If you want to create a new implementation of a library function, place the new function in an object file or your own library. Include your object or library when you link the application. Your version of the function will be used instead of the standard library version.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

3.5 Debugging the application with AXD

AXD enables you to run and debug your ARM-targeted image using any of the following debugging systems:

- ARMulator (the default)
- Multi-ICE
- EmbeddedICE
- Angel debug monitor.

See the *ADS Debuggers Guide* for more information on using the debuggers.

3.5.1 Starting AXD

Start AXD in any of the following ways:

- If you are running under UNIX, either:
  - from any directory type the full path and name of the debugger, for example, `/opt/arm/axd`
  - change to the directory containing the debugger and type its name, for example, `./axd`
- If you are working in the CodeWarrior IDE, open a project and select **Edit** → ***target* Settings...** → **Debugger** → **ARM Debugger** to ensure that AXD is the default debugger and other settings are as you require, then click the **Run/Debug** button or select **Debug** from the **Project** menu.
- If you are running Windows, select **Start** → **Programs** → **ARM Developer Suite 1.1** → **AXD Debugger**.
- If you are using a Windows DOS shell, you can start AXD with the following arguments. Arguments must be in lowercase:

  `-debug` *ImageName*
  : Load *ImageName* for debugging.

  `-exec` *ImageName*
  : Load and run *ImageName*.

  `-logo` Show splash screen (this is the default).

  `-nologo` Suppress splash screen.

  For example, to launch AXD and load `dhryansi.axf` for debugging, type:

  ```
  axd -debug dhryansi.axf
  ```


Copyright © 1999, 2000 ARM Limited. All rights reserved.

Loading an image

If you start AXD from the CodeWarrior IDE, or specify an image name on the DOS command line, an image is already loaded into AXD. If you start AXD without specifying an image, the most recently loaded image is reloaded, if possible. Use **File** → **Load Image** to load a new image (Figure 3-14).

Figure 3-14 Loading an image

Stepping through an application

Use **Execute** → **Step** to step through the application (Figure 3-15).

Copyright © 1999, 2000 ARM Limited. All rights reserved.

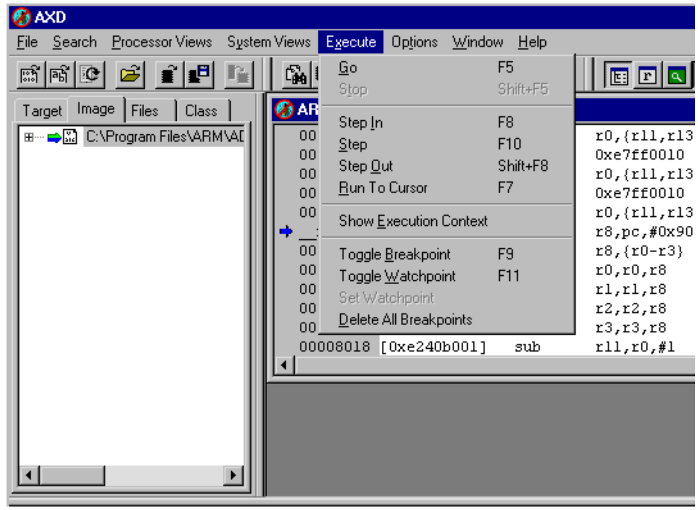

Figure 3-15 The Execute menu

The disassembled code is displayed and a pointer indicates the current position (Figure 3-16). Use **Step** (F10) to execute the next instruction.

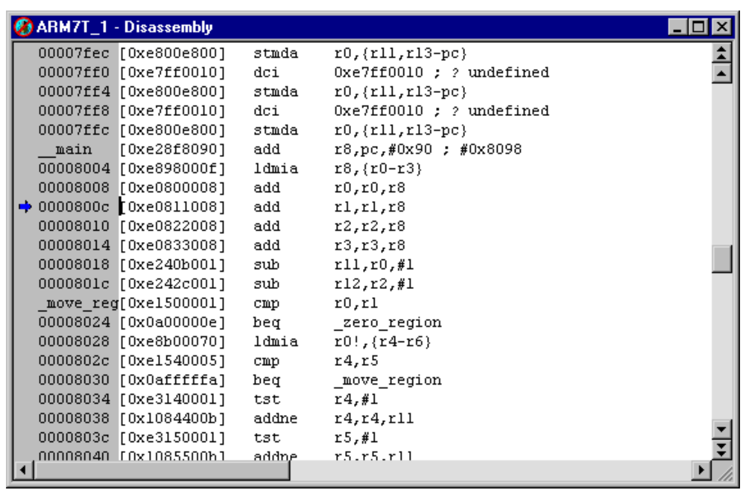

Figure 3-16 Code

Processor view

Use the **Processor Views** menu to monitor the program data during the debug (Figure 3-17).

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

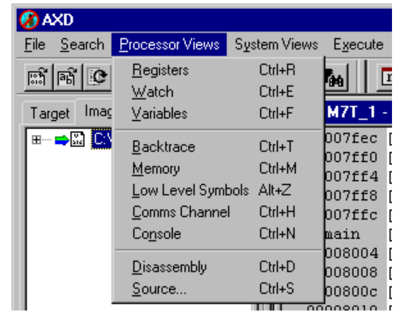

Figure 3-17 Processor Views menu

For example, use **Processor Views** → **Register** to display a dialog showing the register contents (Figure 3-18).

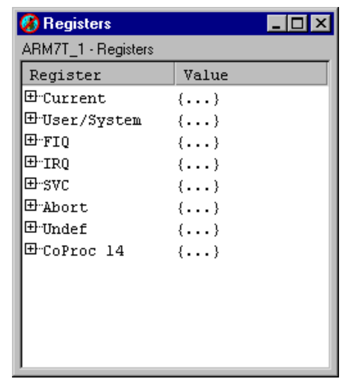

Figure 3-18 Viewing register contents

3.5.2 Configuring ARMulator for AXD

When you run AXD for the first time, an ARMulator debugging session starts by default, with ARMulator configured by settings held in a default configuration file.

For information on reconfiguring ARMulator, returning to ARMulator after using another debug target, and selecting and configuring other debug targets, refer to the *ADS Debuggers Guide*.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Chapter 4
Migrating Projects from SDT to ADS

This chapter describes some of the issues involved when converting an existing project built with the ARM Software Development Toolkit (SDT) to the ARM Developer Suite (ADS). It also shows some of the diagnostic messages which you might see when converting a project, and suggests workarounds for common problems.

It is strongly recommended that you read Chapter 2 *Differences* before reading this chapter.

This chapter contains the following sections:

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.1 Converting makefiles and APM project files

This section describes:

- *Converting APM project files (Windows only)* on page 4-2
- *Converting makefiles (Windows & Unix)* on page 4-3.

4.1.1 Converting APM project files (Windows only)

SDT projects are managed using the ARM Project Manager (APM). ADS projects are managed using the CodeWarrior IDE.

You cannot use existing APM projects, and there is no automatic way to convert APM `.apj` project files to CodeWarrior `.mcp` project files. You must convert APM projects manually.

To convert an APM project to CodeWarrior:

1. Start the ARM Project Manager.
2. Select **APM…** from the **Tools** menu to display the APM preferences panel.
3. Select the **Echo command lines verbosely** checkbox.
4. Rebuild your project. The project log window displays the command line used to invoke each tool.
5. Copy and paste the assemble, compile, and link lines into a temporary text file. For example:

   ```
   [armcc -O1 -echo -W -g+ -MD -DMSC_CLOCK -Ic:\ARM251\INCLUDE]
   ```

   Note

   Do not copy out of the tool configuration windows. The options you can see by opening the window at a certain level in the tree might change further down the tree.

6. Edit the text file to remove the square brackets `[ ]` and any APM-specific options such as `-echo` and `-MD`.

   If there are references to files, such as header files or library files, in the SDT installation directory (for example, ARM251) you might need to change these to point to the ADS installation directory instead.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

7. Check any other assembler, compiler, and linker options displayed on the command line. Some of the defaults have changed. See the appropriate sections in *Moving your development project from SDT to ADS* on page 4-4 for more information on how default compiler, linker, and assembler options have changed between SDT and ADS.

8. Create a new CodeWarrior project. See *Using the CodeWarrior IDE* on page 3-2 for an introduction to CodeWarrior. See the *CodeWarrior IDE Guide* for detailed information.

9. Copy the corrected lines from the text file into the Equivalent Command Line box of the Target Settings dialog for each tool. See *Configuring the project build targets* on page 3-7 for more information.

Alternatively, use the text file as the basis of a makefile. You must edit the text files so that it complies with your makefile format.

4.1.2 Converting makefiles (Windows & Unix)

Some of the assembler, compiler, and linker options have changed. You might need to modify your makefile to account for these changes. See the appropriate sections in *Moving your development project from SDT to ADS* on page 4-4 for more information on how default compiler, linker, and assembler options have changed between SDT and ADS.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

4.2 Moving your development project from SDT to ADS

The following sections describe the most important changes between ADS and SDT, and describe how to change your tool options and code to work with ADS:

- *Compiling* on page 4-4
- *Assembling* on page 4-7
- *Linking* on page 4-8
- *Initialization of C Libraries and Execution Regions* on page 4-13
- *Calling constructors and destructors for top-level C++ objects* on page 4-15.

4.2.1 Compiling

Some compiler features have changed between SDT and ADS. For a full list of changes between SDT and ADS 1.0, see *Changed compiler behavior* on page 2-38. For a full list of changes between ADS 1.0 and ADS 1.1, see *Changes to the compilers and libraries* on page 2-10. The following sections describe changes to the most commonly used compiler options, and to how paths are handled:

- *-apcs 3/nosw* on page 4-4
- *-apcs /softfp/narrow (or /wide)* on page 4-4
- *-zat and alignment of top-level static objects* on page 4-5
- *-zas and alignment of structs* on page 4-5
- *-zz, -zt, -zzt0* on page 4-5
- *-fc* on page 4-6
- *Include paths* on page 4-6

-apcs 3/nosw

The `-apcs 3` options is the default for the compilers in SDT 2.50 and SDT 2.51. It is redundant, and is faulted in ADS.

The `-apcs` qualifier `/nosw` is recognized by the SDT ARM compilers. `/noswst` is the default for ADS.

-apcs /softfp/narrow (or /wide)

`-apcs /softfp` is the default for the compilers in SDT 2.50 and SDT 2.51. The equivalent default ADS option is `-fpu softvfp`. The options are functionally similar. They both implement floating-point arithmetic by subroutine call.

If you see:

```
Error: C3057E: bad option '-apcs /softfp': ignored
```

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

then remove the `/softfp` qualifier from your compiler command line. See *Floating-point support* on page 2-24 for more information.

In ADS, all code is compiled as `-apcs /narrow`. The `-apcs /wide` option was obsolete in SDT 2.50 and SDT 2.51, and is no longer supported.

If you see:

```
Error: C3057E: bad option '-apcs /narrow': ignored
```

then remove the `/narrow` qualifier from your compiler command line. See Table 2-1 on page 2-38 for more information on changed APCS qualifiers.

-zat and alignment of top-level static objects

In SDT, `-zat`*`Number`* specifies the minimum byte alignment for top-level static objects, such as global variables. Valid values for *`Number`* are1, 2, 4, and 8. The default is 4 for the ARM compilers and 1 for the Thumb compilers

The ADS compilers do not support the `-zat` option. The default is the equivalent of `-zat1` for both ARM and Thumb.

-zas and alignment of structs

In SDT, the compiler always places structures on word boundaries (`-zas4`) by default, unless they are packed with the `__packed` qualifier.

The ADS compilers align only as strictly as the contents of the structure require. This is the equivalent of `-zas1`. The `-zas` option is deprecated and the compiler generates the following warning:

```
Warning: C2067I: option -zas will not be supported in future releases
```

It is recommended that you avoid writing code that relies on the alignment of objects such as structures. See the description of structures, unions, enumerations, and bitfields in the *ADS Compiler, Linker, and Utilities Guide* for more information.

To revert to SDT behavior and place structures on word boundaries, compile with `-zas4`.

-zz, -zt, -zzt0

In SDT, the compiler options `-zz0` and `-zt` are commonly combined as `-zzt0`. This forbids the use of tentative declarations, and forces uninitialized globals to be placed directly in the ZI area.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Note

`-zz-1` is a deprecated option that gives the same result as `-zz0`. Use `-zz0` in preference to `-zz-1` in SDT 2.50 and SDT 2.51.

In ADS, `-zz0`, `-zt`, `-zzt0`, and `-zz-1` are faulted. The behavior specified by `-zz0` and `-zt` is the default in ADS unless `-strict` is used. Remove these options from your compiler command line.

This change might affect linking if you are using a scatter-load description file. See *Linking* on page 4-8 for more information.

-fc

In the SDT 2.11a, and earlier toolkits, the `-fc` option enabled *limited pcc* support. This is redundant in SDT 2.50 and SDT 2.51. Using `-fc`:

- allowed dollar characters (`$`) in identifiers
- suppressed warnings on explicit casts between function and object pointers
- allowed junk at the end of preprocessor directive lines.

The first two of these are the default in SDT 2.50, SDT 2.51, and ADS, unless `-strict` is used. To allow junk at the end of preprocessor directives, use the `-Ep` option instead.

For backward compatibility with old projects, the `-fc` option is not faulted in SDT 2.50 and SDT 2.51, but it has no effect over the normal defaults and is not documented.

The ADS compilers fault `-fc`. Remove it from your compiler command line.

Include paths

It is recommended that you use the CodeWarrior IDE Access Paths tab, not the Equivalent Command Line field, to specify compiler *include paths* (such as `'-I.\include')` in a CodeWarrior project.

For example, `'-I..\include'` is the same as the project relative path `'{Project}..\include'`.

Note

It is recommended that you do not use recursive path searching. See the *CodeWarrior IDE Guide* for details.

You can use `-I` in the Equivalent Command Line field if you must follow Berkeley search rules (the default compiler command-line behavior) or K&R search rules, instead of the CodeWarrior behavior. However, CodeWarrior Browser information and

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Error processing is unlikely to work correctly because the `-I` option does not update CodeWarrior internal path information. CodeWarrior cannot find files that the compilers input from paths specified with `-I`. The Access Paths tab also enables you to move a project without moving its source files.

4.2.2 Assembling

Some assembler features have changed between SDT and ADS. For a full list of changes between SDT and ADS 1.0, see *Changed assembler behavior* on page 2-44. For a full list of changes between ADS 1.0 and ADS 1.1, see *Changes to the assembler* on page 2-14. The following sections describe changes that most frequently cause problems when moving to ADS:

- *Interworking* on page 4-7
- *New FUNCTION directive* on page 4-7
- *String Literals and $* on page 4-8.

Interworking

In SDT, assembly language code intended for interworking is marked with the `INTERWORK` attribute on the `AREA` directive. For example:

```
AREA Thumb,CODE,READONLY,INTERWORK
```

In ADS the `INTERWORK` attribute is obsolete and has been replaced with the `/interwork` qualifier to the `-apcs` option. The assembler gives the following warning:

```
INTERWORK area directive is obsolete. Continuing as if -apcs /inter selected.
```

Delete the `INTERWORK` attribute from your assembly language source, and assemble with the `-apcs/interwork` command-line option instead. At link time, the linker adds interworking veneers to the image, if required.

New FUNCTION directive

ADS supports a new ARM assembler directive called `FUNCTION`. If you have any macros in your assembly language code with the name `FUNCTION`, the names will conflict, and the assembly will fail.

You must rename any macros that use the name `FUNCTION`. For example, change the macro:

```
FUNCTION <label>
```

to:

```
FUNCTION1 <label>
```

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

String Literals and $

If you are porting SDT code that contains `$` symbols in strings, (for example, initialization code that performs the RO and RW execution region copying and ZI initialization), you must change `$` to `$$` to build under ADS. For example, change:

```
basesym SETS   "|Image$$":CC:namecp:CC:"$$Base|"
```

to:

```
basesym SETS   "|Image$$$$":CC:namecp:CC:"$$$$Base|"
```

See *Changed assembler behavior* on page 2-44, and the *ADS Assembler Guide* for more information.

Note

In ADS, RO/RW/ZI initialization is usually done by the C library. You might not need your SDT initialization code. See *Initialization of C Libraries and Execution Regions* on page 4-13 for more information.

4.2.3 Linking

Some linker features have changed between SDT and ADS. For a full list of changes between SDT and ADS 1.0, see *Changed linker behavior* on page 2-47. For a full list of changes between ADS 1.0 and ADS 1.1, see *Changes to the linker* on page 2-16. The following sections describe changes that most frequently cause problems when moving to ADS:

- *Specifying libraries* on page 4-8
- *Change -info size to -info sizes* on page 4-9
- *Change -symbols file to -symbols* on page 4-9
- *Linking old objects* on page 4-9
- *Linking old libraries* on page 4-10
- *Unused section elimination* on page 4-10
- *Use of +RW and +ZI in scatter-loading* on page 4-11
- *Generating binary images* on page 4-13.

Specifying libraries

With SDT, it is common to specify C libraries on the linker command line, in particular if you are using the Embedded C libraries.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

With ADS:

- There are no Embedded C libraries supplied with ADS. You can retarget the standard C libraries for embedded use. See the description of tailoring the C library to a new execution environment in the *ADS Compiler, Linker, and Utilities Guide* for detailed information.
- The names of the C libraries are different to those used for SDT. See the description of library naming conventions in the *ADS Compiler, Linker, and Utilities Guide*.
- The linker normally finds the correct C or C++ libraries to link with, and it might use several libraries, so do not specify the C or C++ libraries on the linker command line.

  For example, change:

  ```
  armlink obj1.o obj2.o armlib_cn.32l -o image.axf
  ```

  to:

  ```
  armlink obj1.o obj2.o -o image.axf
  ```

Change -info size to -info sizes

In SDT 2.50 and SDT 2.51, the linker accepts either `size` or `sizes` as a qualifier to the `-info` option. In ADS, only `sizes` is accepted.

Change -symbols file to -symbols

In SDT 2.50 and SDT 2.51, the `-symbols` option requires a filename as a parameter.

In ADS, the `-symbols` option has no parameter. If output to a file is required, use `-list` *`filename`*.

Linking old objects

The object file format for ADS is different to that used by SDT.

In SDT, objects are in the ARM proprietary AOF format. In ADS, the format for all objects and images is the industry standard ELF.

For backwards compatibility, the ADS linker accepts object files in the SDT AOF format and libraries in the SDT ALF format. However, these formats are obsolete and will not be supported in future releases.

Note

The byte order of **`double`** and **`long long`** types has changed.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

In SDT, the formats of little-endian **double** and big-endian **long long** are nonstandard.

The ADS compilers and assembler support industry-standard **double** and **long long** types in both little-endian and big-endian formats. See *Byte order of long long and double* on page 2-25 for more information.

If you try to link an ADS object that uses pure-endian **double** with an SDT object that uses mixed-endian **double**, the linker reports an attribute clash:

```
Error: L6242E: Cannot link object _main.o as its attributes are incompatible
with the image attributes.
```

If possible, it is recommended that you rebuild your entire project, including the old objects, with ADS. However, if you do not have the source code for an object or library, try rebuilding your ADS application code with the `-fpu softfpa` option. See *Object and library compatibility* on page 2-33 for a detailed explanation of how and when you can link old library code. See *Floating-point support* on page 2-24 for more information on changes to the floating-point defaults.

Not all AOF relocations are recognized in ADS. This means that some AOF objects cannot be translated to ELF. The linker faults an attempt to link with an AOF object that cannot be translated:

```
Error : (Fatal) L6027U: Relocation #17 in obj.o (SYMBOL_NAME) has
invalid/unknown type.
```

In this case, you must rebuild the object or library with ADS.

Linking old libraries

The library file format has changed between SDT and ADS. SDT libraries are in the ARM proprietary ALF format. The ADS library format is *ar* and armar replaces armlib as the library manager.

For backwards compatibility, the ADS linker accepts object files in the SDT AOF format and libraries in the SDT ALF format. However, these formats are obsolete and will not be supported in future releases. It is recommended that you rebuild your entire project, including the libraries, with ADS. See *Linking old objects* on page 4-9 for more information.

Unused section elimination

In SDT, the `-noremove` linker option is the default. The linker does not remove unused code or data sections unless instructed to do so with the `-remove` option.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

In ADS, `-remove` is the default. Unused code and data sections are removed by default. Use the `-info unused` option to generate a list of sections that have been removed.

To ensure that important sections, such as the vector table, are not removed you must mark them as an entry point. For example, use the assembler `ENTRY` directive. The linker does not remove sections that are marked as an entry point.

The ADS C library defines an entry point at `__main()`. If you specify additional entry points, and do not explicitly specify an initial entry point with the `-entry` option, the linker cannot determine which entry point to use as the initial entry point and gives a warning:

```
Image does not have an entry point. (Not specified or not set due to multiple
choices)
```

You can select one of the entry points as the initial image entry point. For example, use `-entry 0x0` for ROM images that are entered at reset.

ARM cores that support WinCE have a high vector pin. For example, the ARM920T has the HiVecs pin, so that the vector table can be moved to `0xFFFF0000`. In this case, link with `-entry 0xFFFF0000`.

See the description of image entry points in the *ADS Compiler, Linker, and Utilities Guide* for more information.

Use of +RW and +ZI in scatter-loading

SDT places global variable declarations such is:

```
int a;
```

into the RW data area, unless the compiler switch `-zzt0` is used, in which case it is placed into the ZI data area. See *-zz, -zt, -zzt0* on page 4-5 for more information.

In ADS 1.0 and later an uninitialized global variable is always be placed into the ZI data area. In ADS 1.1 and later, zero initialized global definitions such as:

```
int a=0;
```

are also placed in the ZI data area.

This change in default behavior can cause problems with some SDT scatter-load description files.

Example 4-1 shows a typical scatter-load description file that can be used with SDT, where `int a;` is declared in `periph.c`:

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Example 4-1

```
LOAD_FLASH 0x04000000 0x80000   ; start address and length
{   EXEC_FLASH 0x04000000
    {   init.o (Init,+FIRST)    ; remap & init code
        * (+RO)                 ; all other RO areas
    }
    Peripherals 0x02000000
    {   periph.o (+RW)          ; Variables for accessing
                                ; peripherals
    }
    32bitRAM 0x0000
    {   vectors.o (Vect,+FIRST) ; vector table
        int_handler.o (+RO)     ; interrupt handler
    }
    16bitRAM 0x2000
    {    * (+RW,+ZI)            ; program variables
    }
}
```

If `periph.c` contains only uninitialized global variables, and this scatter-load description file is used with ADS, the linker gives the following warning message:

```
Warning : L6314W: C:\scatter.scf(line 7, col 19) No section matches pattern
periph.o(RW).
```

because the linker cannot identify any RW data from `periph.o` that can be placed into this execution region. In Example 4-1, the ZI data that is produced when compiling `periph.c` is placed into the 16bitRAM execution region by the wildcard placement rule:

```
16bitRAM 0x2000
    {    * (+RW,+ZI)            ; program variables
    }
```

This causes the application to execute incorrectly because accesses to the variables from `periph.c` will no longer map onto the actual peripheral registers.

If the 16bitRAM wildcard region had not been defined, the link would fail because the ZI section generated from `periph.c` would not match any placement rule.

To fix this problem, you must change the specification for the `Peripherals` execution region to:

```
    Peripherals 0x02000000
    {   periph.o (+ZI) ; Variables for accessing peripherals
```

See also *Memory Mapped Peripherals* on page 4-15.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Generating binary images

In SDT 2.50 and SDT 2.51, you can convert the ELF output image from the linker into a plain binary file with:

```
fromelf -nozeropad image.axf -bin image.bin
```

In ADS, the syntax has changed. Use:

```
fromelf image.axf -bin -o image.bin
```

The `-nozeropad` option is redundant in ADS, because fromelf never pads output images with zeros to represent the ZI section.

The SDT 2.50 and SDT 2.51 linkers warn that `-bin` and `-aif -bin` will not be supported in future releases. The ADS linker faults these options. Use fromelf to generate the binary instead.

4.2.4 Initialization of C Libraries and Execution Regions

This section applies to SDT projects that link with the Embedded C libraries (in the `\lib\embedded` directory) to avoid the use of semihosting SWIs. It describes:

- *Application Entry Point* on page 4-13
- *Library Entry Point* on page 4-14
- *RW/RO and ZI Region Initialization* on page 4-14
- *Memory Mapped Peripherals* on page 4-15

Application Entry Point

The SDT embedded C libraries do not require initialization. It is recommended that you use `C_Entry()`, instead of `main()` as the entry point for your C code. This ensures that the full ANSI C semihosting library initialization code is not linked in.

There are no Embedded C libraries supplied with ADS because the standard C libraries can be retargeted for embedded use. However, the standard libraries must be initialized. This is normally done through `main()`.

This means that you must have a `main()` function if you use the C libraries in ADS. If you are moving a project from SDT to ADS, rename `C_Entry()` to `main()`.

See the section on tailoring the C library to a new execution environment in the *ADS Compiler, Linker, and Utilities Guide* for more information on retargeting the C libraries.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

Library Entry Point

With the SDT, you typically call `C_Entry` from some assembler initialization code that initializes, for example, stack pointers.

With ADS, `__main` is the entry point for the C library. Change your initialization code to branch to __main instead of `C_Entry`. For example, change:

```
    BL C_Entry
```

to:

```
    B __main
```

Use a `B` instruction (not `BL`) because an application will never return this way.

RW/RO and ZI Region Initialization

With the SDT, you must write your own code to initialize RW and ZI variables and to relocate RO code to RAM, if required. That is, you must copy the RW and RO data and code from ROM to RAM, and zero the ZI data.

With ADS you do not need to write your own initialization code because the C library code within `__main()`:

1. Copies non-root RW and RO execution regions from their load addresses to their execution addresses.
2. Zeroes ZI regions.
3. Branches to `__rt_entry`, to initialize the library, which ultimately calls your `main()`.

If you are moving from SDT to ADS you can remove or comment out the redundant initialization code.

If you want to continue using your own initialization code to perform RO, RW execution region and ZI initialization, define your own `__main` that branches to `__rt_entry`:

```
    IMPORT __rt_entry
    EXPORT __main
    ENTRY
__main
    B     __rt_entry
```

See the description of how C and C++ programs use the library functions in the C library chapter of the *ADS Compiler, Linker, and Utilities Guide*.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Memory Mapped Peripherals

If you have C variables mapped onto the registers of, for example, memory mapped peripherals, you can instruct the ADS library not to zero-initialize them.

Example 4-2 defines a C structure mapped onto some peripheral registers in a file called `iovar.c`:

Example 4-2 iovar.c

```
struct {
    volatile unsigned reg1; /* timer control */
    volatile unsigned reg2; /* timer value   */
} timer_reg;
```

Example 4-3 adds a root region to the scatter-load description file to place the output from `iovar.c` at the required address in the memory map. The region is labeled `UNINIT` to ensure that the ZI section is not zero-initialized.

Example 4-3 Scatter-load description for iovar.c

```
IO 0x40000000
{
    IO 0x40000000 UNINIT
    {
        iovar.o (+ZI)
    }
}
```

4.2.5 Calling constructors and destructors for top-level C++ objects

If you are using ARM C++ with SDT you must:

- call constructors for top-level C++ library objects with an explicit call to `__cpp_initialise()`
- call destructors for top-level C++ library objects with an explicit call to `__cpp_finalise()`.

This is described in the SDT 2.51 Errata PDF, and in Application Note 74 *Using ARM C++ in Embedded Systems*.

Copyright © 1999, 2000 ARM Limited. All rights reserved.

In ADS:

- constructors for top-level C++ library objects are called by `__rt_lib_init()`.
- destructors for top-level C++ library objects are called by `__rt_lib_shutdown()`.

You must not call `__cpp_initialise()` and `__cpp_finalise()` from your application code.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.

 Copyright © 1999, 2000 ARM Limited. All rights reserved.