

ARM[®] Firmware Suite

Version 1.4

User Guide



ARM Firmware Suite

User Guide

Copyright © 2000-2002 ARM Limited. All rights reserved.

Release Information

Change History

Date	Issue	Change
April 2000	A	New document
June 2001	B	Second version
March 2002	C	Third version
September 2002	D	Interim release for Excalibur support

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

ARM Firmware Suite User Guide

Preface

About this document	vi
Further reading	viii
Feedback	x

Chapter 1

Introduction to the ARM Firmware Suite

1.1	About the ARM Firmware Suite	1-2
1.2	What is firmware?	1-4
1.3	Flash library and utilities	1-7
1.4	Additional libraries	1-11
1.5	µHAL demonstration programs	1-13
1.6	Angel	1-16

Chapter 2

An Introduction to µHAL

2.1	About µHAL	2-2
2.2	Building a new µHAL-based application	2-7
2.3	Building the µHAL library	2-9

Chapter 3

Running the LED Application

3.1	About the LED application	3-2
3.2	Building the LED application	3-4
3.3	Using Multi-ICE to load and debug images	3-7

	3.4	Using Angel to load and debug images	3-13
	3.5	Modifying the application	3-21
Chapter 4		Running the Timer Application	
	4.1	How the application uses μ HAL	4-2
	4.2	Running the application	4-5
Chapter 5		Creating a New Application	
	5.1	How to create a new application	5-2
	5.2	Extending the existing directory structure	5-3
	5.3	Creating a separate directory structure	5-10
Chapter 6		Troubleshooting and Frequently Asked Questions	
	6.1	Frequently asked questions	6-2
	6.2	Troubleshooting	6-5
		Glossary	

Preface

This preface introduces the ARM Firmware Suite and provides a guide to using the software. It contains the following sections:

- *About this document* on page vi
- *Further reading* on page viii
- *Feedback* on page x.

About this document

This book provides a guide on how to setup and use the ARM Firmware Suite. It describes its major components and features. This book contains a simplified guide to downloading and running the demonstration applications and how to develop applications for ARM-based hardware platforms.

Intended audience

This book is written for hardware and software developers to aid the development of ARM-based products and applications. It assumes that you are familiar with ARM architectures and have an understanding of computer hardware. You can find more detailed information in the guides listed under *ARM publications* on page viii.

Using this book

This document is organized into the following chapters:

Chapter 1 *Introduction to the ARM Firmware Suite*

Read this chapter for an introduction to the *ARM Firmware Suite* (AFS). It describes the individual components of AFS.

Chapter 2 *An Introduction to μ HAL*

Read this chapter for a description of the structure and design of the hardware abstraction layer.

Chapter 3 *Running the LED Application*

Read this chapter for information of the Dhrystone application and how to run it using AFS.

Chapter 4 *Running the Timer Application*

Read this chapter for a description of how an application can use the μ HAL APIs.

Chapter 5 *Creating a New Application*

Read this chapter for instructions on how to build a new application that uses AFS.

Chapter 6 *Troubleshooting and Frequently Asked Questions*

Read this chapter for the solution to common problems.

Typographical conventions

The following typographical conventions are used in this book:

<code>typewriter</code>	Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.
<u>typewriter</u>	Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the whole command or option name.
<code>typewriter <i>italic</i></code>	Denotes arguments to commands and functions where the argument is to be replaced by a specific value.
<i>italic</i>	Highlights important notes, introduces special terminology, denotes cross-references, and citations.
bold	Highlights interface elements, such as menu names. Also used for emphasis in descriptive lists and for ARM processor signal names.
<code>typewriter bold</code>	Denotes language keywords when used outside example code.

Further reading

This section lists publications from ARM and third parties that provide additional information about developing on ARM processors.

ARM publications

The following publication provides detailed information about AFS components:

- *ARM Firmware Suite Reference Guide* (ARM DUI 0102).

The following publications provide information about ARM Integrator products:

- *ARM Integrator/CM920T User Guide* (ARM DDI 0097)
- *ARM Integrator/CM940T User Guide* (ARM DDI 0125)
- *ARM Integrator/CM720T User Guide* (ARM DDI 0126)
- *ARM Integrator/CM740T User Guide* (ARM DDI 0124)
- *ARM Integrator/CM7TDMI User Guide* (ARM DDI 0126)
- *ARM Integrator/SP User Guide* (ARM DUI 0099)
- *ARM Integrator/AP User Guide* (ARM DUI 0098).

The following publication provides information about ARM Prospector products:

- *ARM Prospector/P1100 User Guide* (ARM DUI 122).

The following publications provide information about ARM hardware and software debugging tools:

- *RealMonitor Host Controller User Guide* (ARM DUI 0137).
- *RealMonitor Target Controller User Guide* (ARM DUI 0142).
- *Multi-ICE User Guide* (ARM DUI 0048)

The following publications provide information about the ARM Developer Suite:

- *ADS Getting Started* (ARM DUI 0064)
- *ADS Tools Guide* (ARM DUI 0067)
- *ADS Debuggers Guide* (ARM DUI 0066)
- *ADS Debug Target Guide* (ARM DUI 0058)
- *ADS Developer Guide* (ARM DUI 0056)
- *ADS CodeWarrior IDE Guide* (ARM DUI 0065).

Further information can be obtained from the ARM web site at:

<http://www.arm.com>

Other publications

The following publications provide information and guidelines for developing products for Microsoft Windows CE:

HARP Enclosure Requirements for Microsoft® Windows® CE 1998 Microsoft Corporation

Standard Development Board for Microsoft® Windows® CE 1998 Microsoft Corporation.

Further information on Microsoft Windows CE is available from the Microsoft web site:

<http://www.microsoft.com>

The following publication provides information about μ C/OS-II:

- *MicroC/OS-II, The Real-Time Kernel*, Jean Labrosse, R&D Technical Books, ISBN 0-87930-543-6.

Feedback

Feedback on both the ARM Firmware Suite and the documentation is welcome.

Feedback on this book

If you have any comments on this book, please send email to errata@arm.com giving:

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

General suggestions for additions and improvements are also welcome.

Feedback on the ARM Firmware Suite

If you have any problems with the ARM Firmware 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 used, including the version number and date.

Chapter 1

Introduction to the ARM Firmware Suite

The *ARM Firmware Suite* (AFS) is a collection of tools and utilities designed as an aid to developing applications and operating systems on ARM-based systems. This chapter contains the following sections:

- *About the ARM Firmware Suite* on page 1-2
- *What is firmware?* on page 1-4
- *Flash library and utilities* on page 1-7
- *μHAL demonstration programs* on page 1-13.

See the *AFS Reference Guide* for detailed descriptions of AFS components, utilities, and libraries.

1.1 About the ARM Firmware Suite

AFS provides:

μHAL libraries

μHAL (pronounced *Micro-HAL*) is the ARM Hardware Abstraction Layer that is the basis of AFS. μHAL is a set of low-level functions that simplify the porting of operating systems and applications.

Flash library and utilities

The flash library provides an *Application Programming Interface* (API) for programming and reading flash memory. The API provides access to individual blocks or words in flash, and access to images and files. The flash management utilities simplify using flash memory. Use the boot switcher, for example, to select and run one of the images in flash.

Development environment

AFS is an easy to use environment for evaluating ARM-based platforms. The library APIs enable rapid development of applications and device drivers. Reusable code is provided to help develop applications and product architectures on a wide range of ARM and third-party development platforms.

AFS is compatible with the *ARM Development Suite* (ADS). AFS supports use of the Angel debug monitor, Multi-ICE (if the target board supports it), and third-party debug monitors.

Additional components

Additional components provided with AFS include a boot monitor, generic applications, and board-specific applications. Use these components to verify that your development board is working correctly. You can use the source code for the applications as a starting point for your own applications.

Additional libraries

AFS supplies libraries for specialized hardware or exception handling.

- The PCI library supports the PCI bus on the Integrator board.
- A public-domain library for compression, zlib, is included. Refer to the GNU web site for more information on public-domain libraries.
- The Chaining library provides support for exception chaining required, for example, by RealMonitor.
- The VFP library provides support for soft vector floating point routines.

- Angel** A version of Angel that has been implemented using μ HAL is included with AFS.
- μ C/OS-II** μ C/OS-II is a multitasking kernel that is comparable in performance to many commercially available kernels. AFS includes a port of μ C/OS for the ARM architecture using the μ HAL API.

1.2 What is firmware?

Firmware is low-level software that runs on development boards and products. You can use the functions in the firmware directly, or you can use the functions as a starting point for developing your own applications.

One difference between firmware and other code libraries is that the firmware is designed to be development-board and operating-system neutral. AFS uses a *Hardware Abstraction Layer* (μ HAL) to provide a neutral interface to the applications.

AFS provides a collection of standard functions with a known API. This API enables applications to perform hardware-specific operations without requiring a completely new version of the application for each hardware configuration. If the AFS libraries have been ported to the different hardware platforms, you can rebuild the application by recompiling and relinking the application.

AFS includes many example applications as well as flash utilities and low-level libraries that you can build into your application.

Figure 1-1 shows the logical organization of code in a development board and how the AFS firmware simplifies application creation by separating low-level board-specific code from higher-level applications.

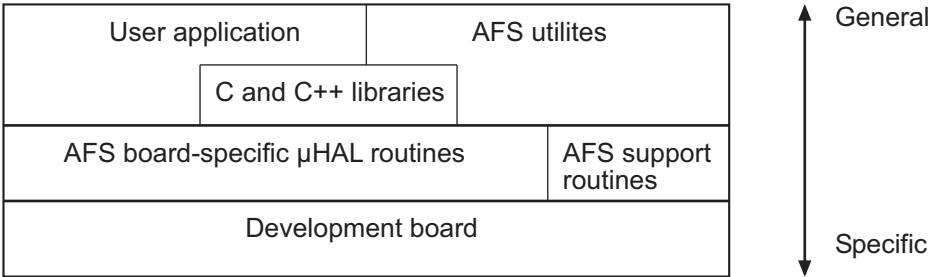


Figure 1-1 Logical organization

1.2.1 μ HAL Libraries

The μ HAL libraries in AFS mask hardware differences between platforms by providing a standard layer of board-dependent functions for the size and location of I/O, RAM, boot flash, and application flash. The μ HAL API provides common and uniform access to other firmware components and applications.

Board and processor-independence for applications is achieved by a set of low-level functions that:

- identify and initialize the system processor or multiple processors

- identify and initialize the system memory
- identify and initialize the system buses, for example PCI
- identify and initialize system devices, for example serial interfaces
- initialize and handle interrupts
- access code or data stored in flash memory.

Examples of specific functional modules in μ HAL are:

- system initialization
- serial ports
- generic timers
- generic LEDs
- interrupt control
- memory management (cache and MMU).

In addition to providing a linkable library, μ HAL also provides a set of definitions (board and processor), and reusable code. Even if you do not use the μ HAL library, you can use the μ HAL definitions in your application.

You can write μ HAL applications to operate in one of two modes:

Standalone A standalone application is one that has complete control of the system from boot time onwards.

Semihosted A semihosted application is one for which an application or debug agent, such as Angel or Multi-ICE, provides or simulates facilities that do not exist on the target system.

The μ HAL applications programming interface is described in the *AFS Reference Guide*.

Directory structure

AFS code and source files are arranged into a directory structure that simplifies identifying the general functional area and the board-specific dependencies.

The `lib` directory, for example, contains prebuilt library archives for different development boards. The directory `AFSv1_4\lib\Integrator` contains generic ARM7TDMI code for use on an Integrator board. The directory `AFSv1_4\lib\Integrator940T` contains code specifically for an Integrator board with an ARM940T core module fitted.

The AFSv1_4\Source\uHAL\Build directory contains build files for different development boards. The directory AFSv1_4\Source\uHAL\Build\Integrator.b contains generic ARM7TDMI code for use on an Integrator board. The directory AFSv1_4\Source\uHAL\Build\Integrator940T.b contains build files specifically for an Integrator board with an ARM940T core module fitted.

1.3 Flash library and utilities

ARM development boards contain flash memory that you can use to store programs and data. You can:

- Use the library functions to access flash from your own application.
- Use the *ARM Flash Utility* (AFU) to load applications into the flash or RAM memory.
- Use the *ARM Boot Flash Utility* (BootFU) application to program the boot and FPGA areas of flash memory.
- Use the boot switcher subprogram, normally located within the first application that is run on reset, to select and run an image in application flash. You can store one or more code images in flash memory and use the boot switcher to start the image at reset.
- Use the boot monitor to load an image from the serial port and select it to run when the development board is reset. The boot monitor also provides a simple command-line interface that provides system debug and self-test functions.

1.3.1 The flash library APIs

The flash library provides four types of function that you can use in your application:

- functions that directly access flash memory
- functions related to low-level file structures
- functions related to high-level file access
- functions related to application-defined storage areas.

The flash management library is described in the *AFS Reference Guide*.

1.3.2 The ARM Flash Utility

The AFU application can manipulate and store data within a system that uses the flash library. The AFU runs within an ARM debug environment such as the ARM Multi-ICE server and the *ARM Extendable Debugger* (AXD). AFU commands are available to:

- list image information
- delete a block of flash
- program an image from the host computer into flash
- read an image from flash and send it to the host computer
- examine a block of flash for problems.

The commands for AFU are described in the *AFS Reference Guide*.

1.3.3 The ARM Boot Flash Utility

With the BootFU application, you can program the boot and FPGA areas of flash memory. BootFU must be loaded into the target system RAM to operate. BootFU runs within an ARM debug environment such as the ARM Multi-ICE server and the AXD environment. This application is only used for upgrading or restoring the boot ROM and is not required for loading and debugging user applications.

1.3.4 Boot switcher

The boot switcher is a small subprogram, normally located within the first application that is run on reset. The boot switcher selects and runs an image in application flash. You can store one or more code images in flash memory and use the boot switcher to start the image at reset.

When the ARM development board is reset, the boot switcher reads the status of a hardware switch and, depending on the value, either:

- Runs the default application. (The default application is typically the boot monitor command interpreter.)
- Searches flash for the image specified in the system information block and runs that image instead of the boot monitor. If the image is not found in flash, an error code is passed to the default application and it displays an error message.

1.3.5 Boot monitor

Use the boot monitor to load an image from the serial port and select it to run when the development board is reset. The boot monitor also provides a simple command-line interface that provides system debug and self-test functions.

The boot monitor communicates with a host computer using simple commands over a serial port. The boot monitor conforms to the *Microsoft Standard Development Board Requirements for Windows CE Specification*. The requirements of the *Microsoft Harp Specification* have been extended by the ARM boot monitor to aid development of new hardware. In particular, new system-specific commands have been added.

See the *AFS Reference Guide* for a detailed description of the boot monitor commands.

1.3.6 Managing images in flash

The flash memory on development boards is logically divided into two areas:

Application Application flash holds data and user applications. You normally load your own programs into the application flash.

Boot Boot flash holds the boot monitor and boot switcher utilities used for loading and debugging applications. This flash can be modified using the BootFU application, but this is not a typical user action.

The images stored in flash memory have three to five parts:

Code and data

The actual code and data for the image.

Header If the original image contained a header, the header is moved to after the end of the image. Not all images have a header.

Image info The Image Information Block holds additional information about the image such as image name and start address.

Empty If the image and related data does not completely fill the flash block, there is an area of empty flash before the footer.

Footer The owner of the image, a checksum, and the image number are stored in the footer.

There are three images in the flash memory map shown in Figure 1-2. One of the images is the boot monitor itself. Two other images have been loaded and can be run from the boot switcher. Two other memory blocks are shown in the figure:

Unused This area can be used to hold additional user images.

SIB A *System Information Block* (SIB) flash block is a nonvolatile storage area for various processes.

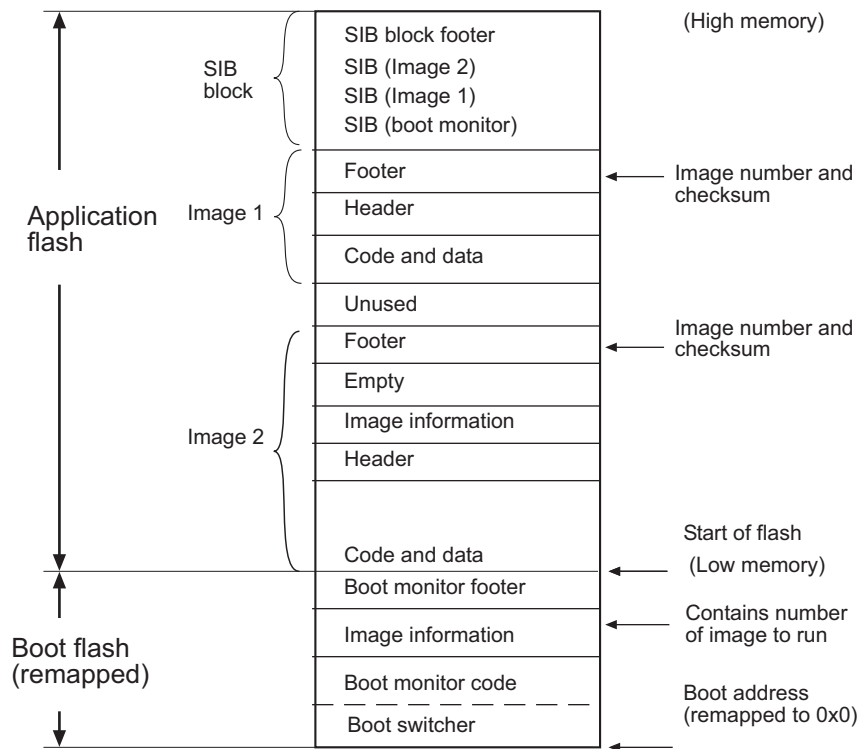


Figure 1-2 Images in flash

1.4 Additional libraries

Additional libraries are provided for managing specialized hardware and supporting exception handling.

1.4.1 PCI library

Some ARM development systems are equipped with PCI expansion card interfaces. Use the PCI library and μ HAL library extensions to initialize and manage a PCI interface and its devices.

The PCI library code has three main functions:

- to initialize the PCI subsystem, that is, to identify the PCI devices and buses in the system and then assign them resources
- to locate PCI devices by device drivers
- to allow the PCI device drivers to control their devices.

Using the PCI management library is described in the *AFS Reference Guide*.

1.4.2 Chaining library

Some hardware and software combinations require that an exception vector, especially an interrupt vector, is shared by different code modules. The chaining library provides a mechanism for installing and updating chains of exception vectors.

The SWI interface is used to obtain information about the debugger in use and to install the trap handler into the chain for a given vector. There are three new SWIs that are used for interrupt chaining:

angel_SWIInfo (0x21)

Return debugger information.

angel_SWIInfo (0x22)

Perform chaining action.

angel_MoveSemihostingVector (0x23)

Move semihosting vector (used with Multi-ICE).

Note

The new SWIs are not implemented in older versions of the development and debug tools.

1.4.3 RealMonitor library

RealMonitor (RM) is a small program that, when integrated into your target application or *Real-Time Operating System* (RTOS), allows you to observe and debug your target with minimal intrusion. A debugger such as AXD that runs on a host computer can connect to the target in order to send commands and retrieve data.

There are two parts to debugging using RM:

- A debug host communication mechanism operates between the debugger (such as AXD) and RM. The debugger uses version 1.51rt of the *Remote Debug Interface* (RDI). RM runs on the debug target using the RealMonitor protocol over the DCC. For details on debugging an RM-integrated application from the host, see the *RealMonitor Host Controller User Guide*.
- RM itself, with RT-aware EmbeddedICE logic, runs on the target hardware. It receives, and responds to, commands over the DCC. Some of these commands cause the current processor state to be saved. Depending on the command received, the application can be suspended while interrupt-driven code continues to execute. For more details on the RM target code, see the *RealMonitor Target Controller User Guide*.

These two functional parts are joined using the RM protocol. This provides the framework for sending packets of data between a host machine and a target processor. The RM protocol is carried over a highly reliable DCC, or any word-based transport mechanism, that keeps target-side RM messages as simple as possible.

You must build the RM target library and link it with your application before you perform any integration. The RM build options provide full control over which features you include in any given build. The build options you set are applicable to either C language code, assembly language code, or both.

1.4.4 The zlib library

The zlib library is included to show how third-party libraries can be used with AFS. Refer to the GNU documentation for more information on zlib.

1.5 μ HAL demonstration programs

Use the generic μ HAL demonstration programs on the AFS CD to understand how to use the μ HAL API. Use the test programs to verify that μ HAL has been successfully ported to a system. The generic programs include:

- *Simple tests*
- *Timing tests* on page 1-14.

1.5.1 Simple tests

These demonstration programs are typically the first images you might run on a new target. The following simple tests demonstrate and verify a specific functionality:

<code>hello.c</code>	This program outputs data to the serial port.
<code>io.c</code>	This program takes data input on the serial port and echoes it on the output.
<code>led.c</code>	This program flashes the LEDs in a binary pattern. It requires no additional functionality (such as serial ports) to be working in order to run.

———— **Note** ————

The semihosted version prints out a banner and description on the debugger console, because this functionality is known to be available in semihosted mode.

<code>heap.c</code>	This program allocates and then frees some memory.
<code>simple-caches.c</code>	<p>This program gives an example of simple cache (Data and Instruction) usage that:</p> <ul style="list-style-type: none"> • reads the cache and MMU state • resets the cache and MMU • turns caches and MMU on and off • restores the original state.
<code>system-timer.c</code>	<p>This program combines serial output, LED flashing, timers, and interrupts. This program uses most of the features of μHAL and is a good indicator that a target is functional.</p>

`file-io.c` A file I/O program. It performs file functions on the host and returns the size of a specified file.

Note

This program is only useful when built semihosted and linked with the ADS C Library.

`exception.c`

This program shows how the ADS C Library handles a divide-by-zero exception.

Note

The compiler displays a warning when building this program.

1.5.2 Timing tests

These demonstration programs are computation and memory-intensive, and demonstrate how the performance of applications is affected by caching strategies. The tests are:

`bubble.c` This program sorts a list by comparing each adjacent pair of items in a list in turn, swapping the items if necessary, and repeating the pass through the list until no swaps are required.

`queens.c` This is one of the benchmark programs that you can use to measure performance with different caching strategy. The problem of the eight queens is a well known example of the use of recursion and backtracking algorithms. The program calculates how 20 queens can be placed on a 20 by 20 chess board so that no queens check against any other queen.

`sieve.c` This is an implementation of the Sieve of Eratosthenes algorithm to find all prime numbers up to a certain N . Begin with an (unmarked) array of integers from 2 to N . The first unmarked integer, 2, is the first prime. Mark every multiple of this prime. Repeatedly take the next unmarked integer as the next prime and mark every multiple of the prime. Finally, the unmarked integers are primes.

1.5.3 Board-specific demonstration programs

The AFS distribution CD contains demonstration program sources and images for a specific board and processor combination. The sources enable you to bring new hardware into operation quickly and to gain experience with building applications that use AFS.

Each development board has its own collection of demonstration programs. The Integrator demos are in AFSv1_4\Demos\Integrator. Examples of the programs available for Integrator are:

- | | |
|-----------|--|
| TestSuite | A collection of test routines for the LEDs, serial ports, keyboard, mouse, interrupt handlers, and timers. |
| scanpci | Scans the Integrator PCI bus and shows what PCI devices are connected. |

1.6 Angel

The Angel debug monitor is an application that allows you to develop and debug applications on ARM-based systems. Angel can be used to debug applications running in either ARM or Thumb state.

A typical Angel system has two main components that communicate through a physical link, such as a serial cable:

Debugger The debugger runs on the host computer. It gives instructions to Angel and displays the results obtained from it. All ARM debuggers support Angel, and you can use any other debugging tool that supports the communications protocol used by Angel.

Angel Debug Monitor

The Angel debug monitor runs alongside the application being debugged on the development board.

The internal operation of Angel is described in the *AFS Reference Guide*.

Chapter 2

An Introduction to μ HAL

This chapter describes μ HAL and how it conceals hardware differences between different ARM-based development systems. This chapter contains the following sections:

- *About μ HAL* on page 2-2
- *Building a new μ HAL-based application* on page 2-7
- *Building the μ HAL library* on page 2-9.

2.1 About μ HAL

μ HAL consists of a low-level interface that provides a common set of functions for different ARM-based systems.

In addition to providing a linkable library, μ HAL also provides a set of definitions (board and processor) and reusable code. All the AFS components are built with the μ HAL libraries. Even if you do not use the μ HAL library, you can use the μ HAL definitions in your application.

μ HAL simplifies building applications for development boards by providing a standard layer of board-dependent functions to manage I/O, RAM, boot flash, and application flash. Figure 2-1 shows a block diagram of a development platform.

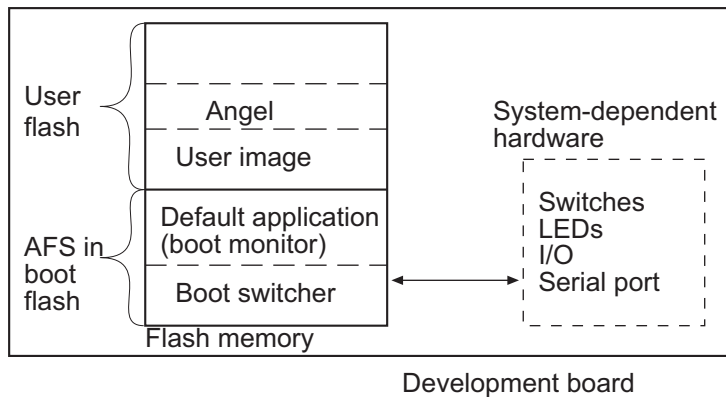


Figure 2-1 Development board with AFS

2.1.1 Licensing

μ HAL and its demonstration programs are freely reusable and you can redistribute them as long as they are used on ARM-based platforms. The other AFS components must be licensed from ARM. See the license agreement on the AFS CD for details of which components are licensed.

If you want to use μ HAL in commercial projects, contact your vendor to ensure that you have the appropriate μ HAL version. Because the other AFS base-level components are not free, you must be careful which parts are used and where.

2.1.2 Frequently asked questions

You can find the answers to some common setup problems and other frequently asked questions in Chapter 6 *Troubleshooting and Frequently Asked Questions*.

The ARM Technical Support pages on the ARM web site have additional information on AFS and on other ARM products:
<http://www.arm.com/DevSupp/Sales+Support/faq.html>

2.1.3 Application programming interfaces

The μ HAL API consists of two types of functions:

- | | |
|-----------------|--|
| Simple | The basic functions allow you to program an application through a minimal number of calls. |
| Extended | The extended functions allow a more complex usage of the system, but you must be aware of the way μ HAL components fit together and how it works. Within μ HAL itself, the basic functions are built using the extended functions. |

For a complete list and description of these functions, see the *AFS Reference Guide*.

2.1.4 Application operating modes

You can write μ HAL applications to operate in one of two modes:

- | | |
|-------------------|--|
| Standalone | A standalone application is one that has complete control of the system from boot time onwards. |
| Semihosted | A semihosted application is one for which an application or debug agent, such as Angel or Multi-ICE, provides or simulates facilities that do not exist on the target system. In the case of a debug agent, access to these facilities are requested by using <i>Software Interrupt</i> instructions (SWIs). |

For example, the serial interface code in a standalone application requires a real serial port to transmit characters. A semihosted application does not necessarily require a serial port of its own to transmit characters, because it uses a SWI to access a communications channel provided by the debug agent.

2.1.5 System support provided by μ HAL

μ HAL provides system support for a variety of target platforms. It does this by providing routines for specific functional modules. These functions are described in the following sections:

- *System initialization software*
- *Serial port*
- *Generic timer*
- *Generic LED* on page 2-4

- *Interrupt control* on page 2-4
- *Memory management* on page 2-5.

System initialization software

This software initializes the system so that the standard μ HAL API is supported. This might involve:

- switching the memory map over from its initial state to the normal state (for example from ROM mapped to physical address 0x00000000 to RAM mapped to virtual address 0x00000000)
- building page tables and setting up memory management or memory protection units
- enabling virtual memory
- setting up the stack
- zeroing user-memory areas.

The system initialization code can be much simpler if the system has already been set up by a debug agent such as Angel.

Serial port

If the system has one or more serial ports, μ HAL allows character-level polled reading and writing to that device by way of low-level C routines or C macros. It also includes a very basic `printf()` implementation. In semihosted mode, μ HAL makes SWI calls to the debug host to run `printf()`.

Generic timer

μ HAL provides a generic interface to manage any timers present in the system. In semihosted mode, μ HAL ensures that it does not use timers that are being used by a memory-resident debug agent such as Angel.

Generic LED

If your system has LEDs, μ HAL provides a generic interface to these LEDs.

Interrupt control

μ HAL supports interrupts that use *Interrupt Requests* (IRQs). This support includes:

- support for applications that request control of an interrupt

- interrupt handling
- interrupt enabling and disabling.

μ HAL assumes that it has complete control of the IRQs. If the application is semihosted, Angelt can use the *Fast Interrupt Requests* (FIQs). Alternatively, use Multi-ICE, which does not require an interrupt.

Note

If you use the chaining library in your application and perform the calls to initialize exception handling, it is possible for the IRQs to be shared by more than one application.

Memory management

If the system has not been set up by a debug agent, μ HAL attempts to set a one-to-one mapping between physical and virtual memory. The one exception is that memory at address 0 must be in RAM.

On some systems, ROM is at address 0. On this type of system, RAM is remapped to address 0, ROM is remapped to address 64M. Because of the remapping, the *Memory Management Unit* (MMU) cannot be disabled or reset.

Note

The memory management and ROM/RAM memory map for the Prospector development board is different from the Integrator board. Refer to the board-specific appendices in the *AFS Reference Guide* for more details.

2.1.6 μ HAL naming conventions

Every μ HAL routine has the following naming conventions:

- the prefix `uHAL`
- the object type
- underscore
- a meaningful capitalized name.

Table 2-1 lists these conventions by object type.

Table 2-1 μHAL naming conventions

Object type	Sample
Basic routine, part of the API	uHALr_GlobalRoutine
Extended routine, part of the API	uHALir_InternalRoutine
Global variable, part of the exported API	uHALv_GlobalVariable
Internal variable, not part of the exported API	uHALiv_InternalVariable
Pointer	uHALp_PointerVariable
Internal pointer	uHALip_InternalPointer
Global structure, part of the exported API	uHALs_GlobalStructure
Global enumerated variable	uHALe_GlobalEnum

Caution

The extended routines provide access to lower-levels of the library functionality. You must have an understanding of how μHAL operates internally before you use the extended routines.

2.2 Building a new μ HAL-based application

There are three ways to build a μ HAL-based application:

- Take an existing demonstration program (for example AFSv1_4/Source/ μ HALDemos/Sources/hello.c) and modify it for your use. You can use the program with any of the existing build systems.
- Create new source files and use one of the build systems such as make or the CodeWarrior IDE. You must understand the details of the build system that you are using.
- Copy the prebuilt μ HAL library that you wish to use, together with the include files (bits.h, cdefs.h, platform.h, sizes.h, and uhal.h) into a new directory and build your application there. However, you must make sure that the μ HAL library is the right variant (board, processor, semihosted versus standalone) and that platform.h is for the system that you wish to run on. This approach is suited to small applications. Determine the required files by viewing the project files or makefiles for an existing application.

The directory names within the Build subdirectory indicate the development board and processor combination. Use the one that matches your development board and target processor.

For example, a directory called Integrator940T.b is for an Integrator board with an ARM940T processor. The directory called Integrator.b is for an Integrator board with a generic ARM7TDMI processor.

Note

If you are using PCI or Flash functions in your application, your makefile must include a path to a prebuilt PCI or Flash library. Prebuilt .a library archives are located in the subdirectory in AFSv1_4\lib\ that corresponds to your development board and processor.

2.2.1 Building with GNUmake

To build μ HAL and its associated components using makefiles use GNUmake. GNUmake is available for UNIX, Linux, and for most Windows versions.

Note

Before you can use GNUmake with Windows 95, Windows 98, or Windows NT, you must first install CygWin. For more information about the CygWin project, it is recommended that you contact Redhat at: <http://sources.Redhat.com>

Installing GNUmake on Unix

To use GNUmake on a Unix workstation, you must:

- have access to the appropriate versions of the ARM toolset (currently ADS v1.0 or higher)
- use the correct version of the build tools and the ARM library
- have an environment variable called ARMLIB that contains a pointer to the set of ARM C libraries
- have an environment variable called ARMINC that contains a pointer to the set of include files
- place gnumake in your search path.

Installing GNUmake on Windows

After installing GNUmake on your system, you must set up some links and environment variables in order to use GNUmake and other Unix tools. Assuming that you are using the bash shell (a popular free command shell available from Cygnus), do the following:

1. Create a desktop shortcut to point to the cygnus.bat file. By default this is placed at c:\cygnus\cygwin-b20\cygnus.bat.
2. Start a bash shell by double-clicking on the desktop shortcut.
3. Move to the directory where the binary files are kept. This directory is quoted in cygnus.bat:

```
bash2-02$ cd //c/cygnus/cygwin-b20/H-i586-cygwin32/bin
```
4. Create a soft link:

```
bash2-02$ ln -s make.exe gnumake.exe
```

The default name for GNUmake is make, however the rules files assume that it is gnumake to avoid conflicts with the native make on Unix systems. Creating a soft link to the real make executable resolves the names.
5. Test that the ARM tools work by launching a new bash shell and typing armcc. armcc executes and lists its input options.
6. Move to the directory of the particular type of board you are building for. (Some components are not board-specific and the makefile exists in the higher-level directory.)
7. Type the command gnumake to build images that contain no debug information, or type gnumake DEBUG=1 to build images that can be debugged.

2.3 Building the μ HAL library

There are two ways of building the μ HAL library and other AFS components. Use either:

- ARM .mcp project files for the CodeWarrior IDE with ADS 1.0 or higher
- GNU makefiles.

Each board and processor combination has its own build directory within AFSv1_4\Source\ μ HAL\Build. Build directories are denoted by the .b suffix on the directory name.

For example, the build directory for the Integrator/CM940 core module is AFSv1_4\Source\ μ HAL\Build\Integrator940T.b, and the build directory for the Integrator with an ARM7TDMI core and Thumb interworking is AFSv1_4\Source\ μ HAL\Build\IntegratorT.b.

The makefile in Integrator940T.b builds two variants of the μ HAL library for the ARM940T Integrator core modules:

- standalone in the subdirectory standalone
- semihosted in the subdirectory semihosted.

Note

The CodeWarrior IDE creates a new output directory called μ HALLibrary_Data and creates the standalone and semihosted subdirectories in the new directory. You must run the CodeWarrior IDE build process twice to build both variants.

See the *AFS Reference Guide* for more information on rebuilding components.

Chapter 3

Running the LED Application

This chapter describes how to build and run the LED demo application. It contains the following sections:

- *About the LED application* on page 3-2
- *Building the LED application* on page 3-4
- *Using Multi-ICE to load and debug images* on page 3-7
- *Using Angel to load and debug images* on page 3-13
- *Modifying the application* on page 3-21.

Note

The examples in this chapter use the Integrator development board. The process is similar for other development boards.

3.1 About the LED application

The LED application flashes the LEDs on the development board in sequence. This is a good first application to test because the code is relatively simple and the LEDs give immediate feedback that the application is working correctly.

3.1.1 Files used with the LED application

AFSv1_4\Source\µHALDemos\Sources\led.c is the single C source file for the LED application in standalone mode. (If the application is semihosted, the file pr_head.c is used. Functions in this file display information about the target on the debug console.)

There are several library files and include files that produce an image for a specific board and processor combination.

The led.c file

The main() function in the C file contains a loop that sequences through the LEDs and turns them off or on. To enable the application to be built for any development board, the µHAL library functions uHALr_InitLEDs() and uHALr_WriteLED() handle the low-level control of the LEDs on the development board.

The LED application does not use µHAL timer or interrupt functions. The delay between LED flashes is determined by a simple loop:

```
for (wait = 0; wait < 1000000; wait++)
```

The platform file

The Include directory contains subdirectories for each of the development boards available. These directories contain the platform.h include file that contains definitions particular to each specific board. For example, the platform.h file for the Integrator board includes #define uHAL_PCI 1. This indicates to the build tools that the Integrator board has PCI slots and supports the µHAL PCI library.

The µHAL library

The uHALDemo applications use µHAL library functions. If the µHAL library cannot be found, the build tools (makefile or project files) build the library.

3.1.2 Overview of the µHAL LED functions

The µHAL function uHALr_InitLEDs() returns the number of LEDs on the development board. The LED application uses this function to set the count variable:

```
count = uHALr_InitLEDs();
```

The function `uHALr_WriteLED()` takes two parameters, the LED number and the state for that LED. The application uses this function to sequence through the LEDs as shown in Example 3-1.

Example 3-1 Flashing the LEDs

```
// Do a binary count on the LEDs
for (i = 0; i < max; i++)
{
    on = (max - 1) & i; // which LEDs are on?
    for (j = 0; j < count; j++)
        uHALr_WriteLED(j + 1, (on & (1 << j) ? 1 : 0));
    for (wait = 0; wait < 1000000; wait++) // wait a while
        ;
}
```

3.1.3 Selecting semihosted or standalone operation

If the application is built as a semihosted image and run with a debugger, the application can use semihosting functions to use resources on the host computer. For example, the debugger console on the host computer is used for printing. The `printf()` function is, therefore, always supported when running semihosted. (The `printf()` and `uHALr_printf()` functions normally map to the same function when an application is built semihosted.)

The semihosted version of the LED application uses the `uHALr_printf()` function to display a message on the debugger console as shown in Example 3-2.

Example 3-2 Printing in semihosted version

```
#ifdef SEMIHOSTED
// init the library
uHALr_LibraryInit();
print_header();
uHALr_printf("\nCheck target for flashing LEDs\n");
#endif
```

3.2 Building the LED application

You must compile and link the LED application before you can download it to your development board. There are two ways to build the image:

- You can use the supplied makefile and build all of the applications. See *Using the makefile*.
- You can use the supplied project file and build each application individually. See *Using the project file* on page 3-4.

After you have built the LED image, it can be downloaded to the development board in any of the following ways:

- If the image is built standalone, convert the .axf file produced by the build system into an .m32 file and download it to flash using the boot monitor. See *Changing the image format* on page 3-6.
- If the image is built standalone, use a debugger and the flash utility to download the .axf file to flash.
- If the image is built semihosted, use a debugger to download the .axf file to RAM.

3.2.1 Using the makefile

The file `makefile` in the board and processor subdirectory builds the demo applications for that development environment. For example,

- `Source\uHALDemos\Build\Integrator940T.b\makefile` builds all of the demonstration applications for an Integrator board with an ARM940T processor
- `Source\uHALDemos\Build\Integrator.b\makefile` builds applications for an Integrator board with a generic ARM7TDMI core module.

Use the make utility to build the applications. If you are using Windows, you must have the GNU make utility installed.

3.2.2 Using the project file

In the board and processor subdirectories, most of the demonstration applications have an ADS project file that builds the application for that development environment. The LED demonstration application, for example, can be built using:

- `AF5v1_4\Source\uHALDemos\Build\Integrator940T.b\led.mcp` with ADS and the CodeWarrior IDE to build for an Integrator board with an ARM940T processor

- AFSv1_4\Source\uHALDemos\Build\Integrator.b\led.mcp builds for an Integrator board with a generic ARM7TDMI processor.

Double-click on the project file or start the development environment and open the project file. The CodeWarrior IDE project window displays the files used to build the application. See Figure 3-1.

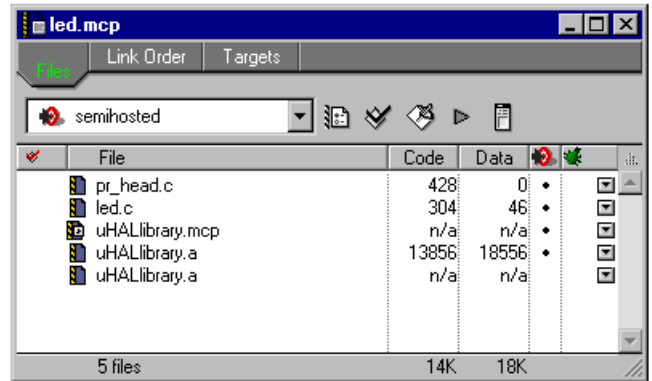


Figure 3-1 Files in the LED project

Select semihosted or standalone as the output target. For ADS and CodeWarrior, the led.mcp file creates a new subdirectory called led_Data that contains the ELF image. Select **Project** → **Make** to rebuild the application. See Figure 3-2.

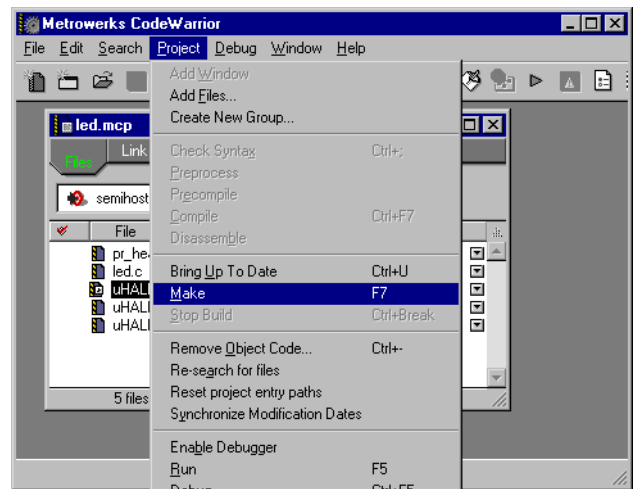


Figure 3-2 Project menu

3.2.3 Changing the image format

If you are using a debugger and the flash utility, you can download .axf files. The boot monitor, however, can only download .m32 Motorola hex files. Use the fromELF utility provided with ADS to convert an .axf file into an .m32 file:

```
fromelf -m32 -output led.m32 led.axf
```

———— **Note** —————

Some of the project files, for example the CodeWarrior project for Angel on Integrator, are set up to perform the conversion to .m32 as part of the build process.

—————

3.3 Using Multi-ICE to load and debug images

If your development board supports debugging over a JTAG interface, and you have the Multi-ICE interface, you can load and debug applications using Multi-ICE.

3.3.1 Preparing the board

Follow the steps below to prepare the target board and Multi-ICE server:

1. Install the Multi-ICE software on your PC.
2. Turn the power to the development board OFF.
3. Connect the Multi-ICE interface cable to the JTAG connector on the development board and connect Multi-ICE parallel cable to the parallel port on the PC.
4. Power-on the development board.
5. Start the Multi-ICE server. See Figure 3-3.

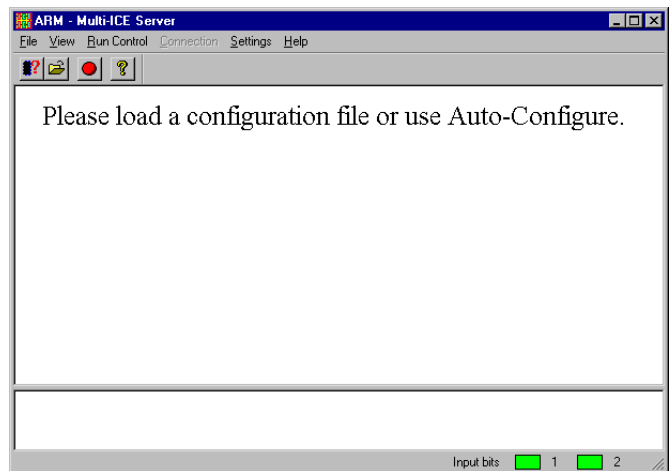


Figure 3-3 Multi-ICE startup

6. Select **Auto configure** to initialize the server and target board. See Figure 3-4 on page 3-8.

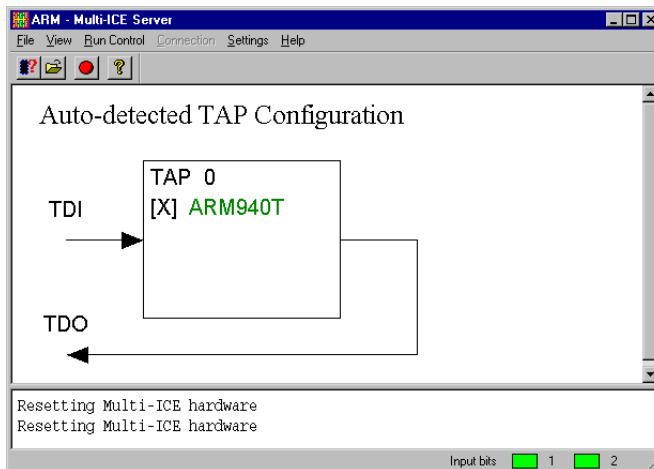


Figure 3-4 Auto configure

3.3.2 Starting the debugger

The boot monitor commands are not available from Multi-ICE, but you can use the flash utility afu to access the flash memory. Follow the steps below to start the debugger and communicate with the Multi-ICE server:

1. Start the debugger (ADW or AXD).
2. Configure the debugger to use Multi-ICE. See Figure 3-5. For more details on configuration, refer to the Multi-ICE documentation.

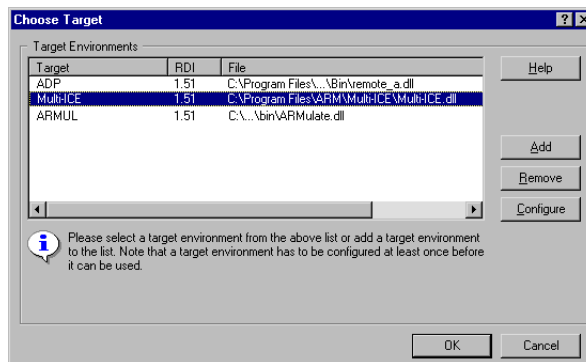


Figure 3-5 Select Multi-ICE

3. If you have not used Multi-ICE with the debugger before, you must use **Add** to install the Multi-ICE .dll file and then **Configure** to initialize the interface.

- Follow the instructions in *Loading an image into flash with Multi-ICE* on page 3-10 to download a standalone image into flash.

3.3.3 Loading an image into RAM with Multi-ICE

If your development board has a JTAG interface and the processor supports Multi-ICE, you can use Multi-ICE with your debugger. Using Multi-ICE and JTAG is much faster than using a serial port.

Note

If your board does not support Multi-ICE, or you do not have the Multi-ICE hardware, you must have the Angel debug monitor in the target board to load and debug applications in RAM. See *Using Angel to load and debug images* on page 3-13.

Connect the development board to the host and establish communication with a debugger:

- Start the debugger as described in *Starting the debugger* on page 3-8.
- Load the semihosted version of the `led.axf` image. See Figure 3-6. (For AXD, select **File** → **Load image**)

Use the image that matches your board. (For the Integrator board with a 940T processor, use the image in

`AFSv1_4\Source\uHALDemos\Build\Integrator940T.b\semihosted`

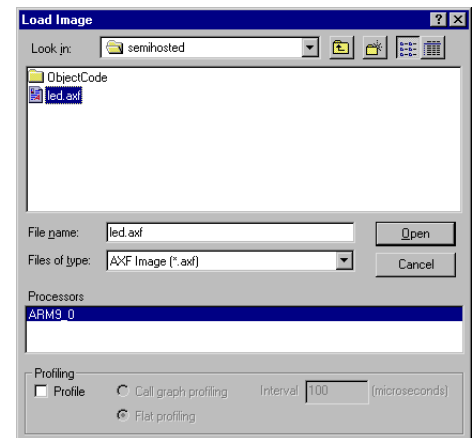


Figure 3-6 Loading the LED application

- Display the debugger console. (For AXD, select **Processor Views** → **Console**.)
- Execute the application. (For AXD, select **Execute** → **Go**.)

Prompt messages are displayed on the screen and the LEDs begin to flash (see Figure 3-7).

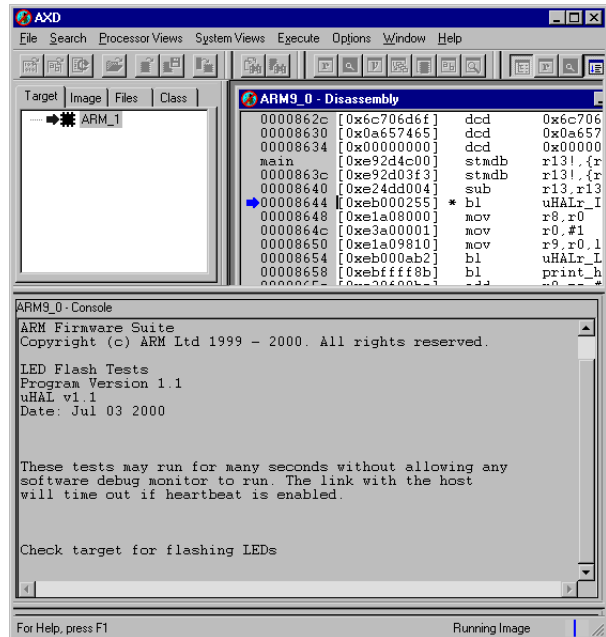


Figure 3-7 Semihosted LED application

3.3.4 Loading an image into flash with Multi-ICE

Follow the steps below to load an image into flash memory on the target board:

1. Start the debugger as described in *Starting the debugger* on page 3-8.
2. Start the flash utility application.
 - a. Use the debugger to load the afu.axf flash utility application (see Figure 3-8 on page 3-11). For AXD, select **File** → **Load image**. Use the image that matches your board. (For the Integrator board, use the image in AFSv1_4\Images\Integrator\.)

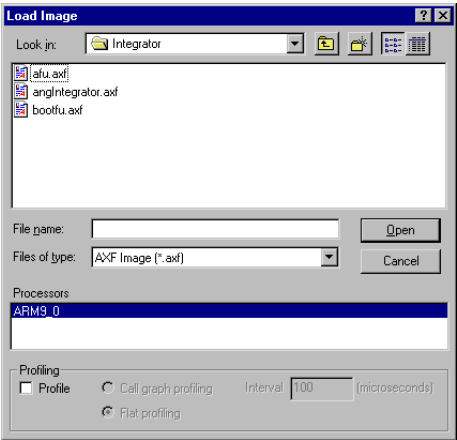


Figure 3-8 Loading the flash utility image

- b. Display the debugger console. (For AXD, select **Processor Views** → **Console**.)
 - c. Execute the flash utility application. (For AXD, select **Execute** → **Go**.)
- The flash utility runs and displays a prompt (see Figure 3-9).

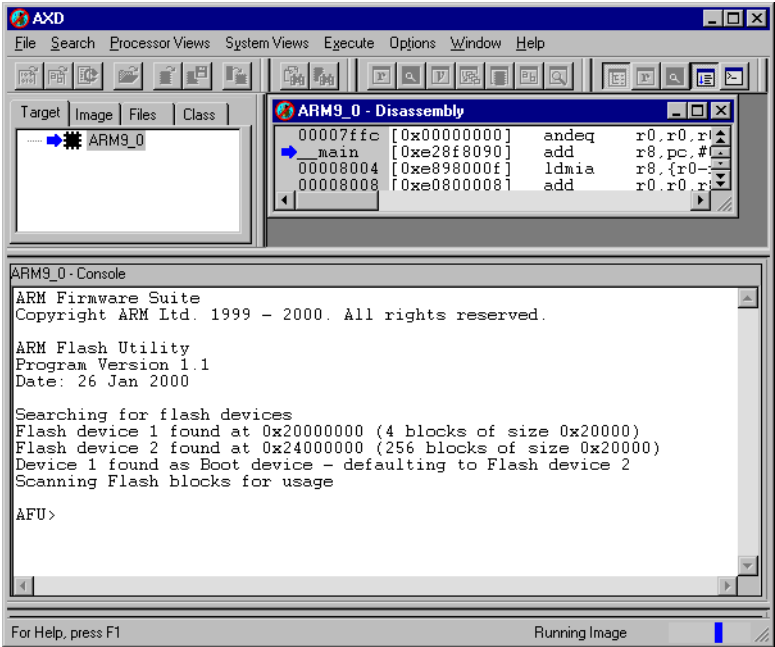


Figure 3-9 ARM Flash Utility

3. Use the Program command to load a standalone image as image number 2. The AFU utility cannot handle long pathnames. To avoid long pathnames, use a temporary directory to hold the files to download:
`p 2 LED D:\Temp\led.axf`
4. Quit by using the `q` command in the debugger and exit the debugger.
5. Set the development board to boot the new image.
 - a. The new image must be set by the boot monitor as the default. Change the development board switches to start the boot monitor on reset. (For the Integrator board, set switch S1-1 and S1-4 to the ON position.)
 - b. Connect the serial cable between the target board and the host and start a terminal emulator. See *Using a serial port and the boot monitor to load and run images* on page 3-17 for terminal settings.
 - c. Reset the development board. You now see the boot monitor prompt, similar to that shown in Figure 3-10.

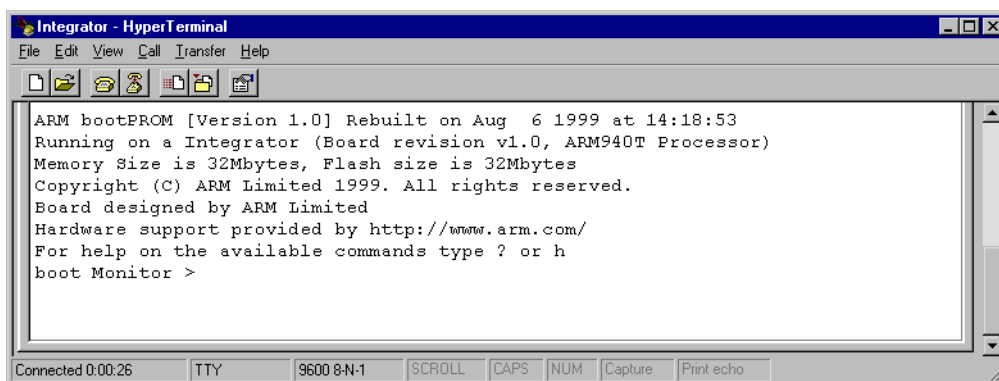


Figure 3-10 Terminal prompt

- d. Enter `BI 2` in the terminal window to select image 2 on reset.
- e. Change the development board switches to run the image selected by the boot monitor. (The settings depend on the board type and version. For the Integrator board, set switches S1-1 to the ON position and S1-4 to the OFF position.)

Note

The switch settings provided here might not apply to the version of development board that you have. Refer to the documentation provided with your development board for detailed switch settings.

6. Reset the board. The LEDs begin flashing.

3.4 Using Angel to load and debug images

If your board does not support Multi-ICE or you do not have the Multi-ICE hardware, use the Angel debug monitor to load and debug images. Some boards come preinstalled with Angel. If your board does not already have Angel installed, you must use one of the utility applications to install Angel. If Angel is installed, you can load and debug the application using the debugger (for example, AXD) and the Angel debug agent.

You can use the serial port on the development board to:

- load an image directly into RAM
- load the flash utility into RAM and start it from the debugger to load a new image into the flash memory
- use the boot monitor to load a new image into flash.

3.4.1 Verifying Angel is in flash

If you are using the serial port and Angel to debug applications, you must have Angel in the target board. There are the following possibilities for Angel in your development board:

No Angel If you do not have an ARM supplied development board, you must build Angel for your board and download it. If you have an older version of an ARM development board, it might not have Angel preinstalled. You must select the appropriate Angel image and use either the boot monitor, or AFU to download it to your board.

Image 0 If you have an older version of an ARM development board, or you have downloaded a customized version of Angel, Angel might be installed as image number 0. Use the boot monitor BI command to select image 0 to run on startup and set the switches on the development board to run the selected image.

Image 911 If you have a newer version of an ARM development board, it might have Angel preinstalled in the system ROM as image number 911. Use the boot monitor BI command to select image 911 to run on startup. (For the Integrator board, set switch S1-1 to the ON position and S1-4 to the ON position to start the boot monitor on reset.)

After you have selected the image, set the switches on the development board to run the selected image, in this case Angel, on startup. (For the Integrator board, set switch S1-1 to the ON position and S1-4 to the OFF position to start the image selected to run on reset.)

Note

The switch settings provided here are a suggestion, but might not apply to the version of development board that you have. Refer to the documentation provided with your development board for detailed switch settings.

3.4.2 Using a terminal to detect Angel

For some development boards, you might need to download the Angel agent before you can use your debugger. To test whether Angel is installed on your board:

1. Set the terminal emulator to 9600 baud (the default Angel baud rate).
2. Use the boot monitor BI command to set the startup image and set the configuration switches to boot the Angel image on startup. (Angel might be installed as either image 0 or image 911. If you have a newer board, it is installed as image 911.)
3. Apply power to the development board and reset the board.
4. Angel attempts to communicate with the debugger over the serial port. The terminal emulator displays some symbols and then the Angel banner. The display will be similar to, but not necessarily the same as, that shown in Figure 3-11.

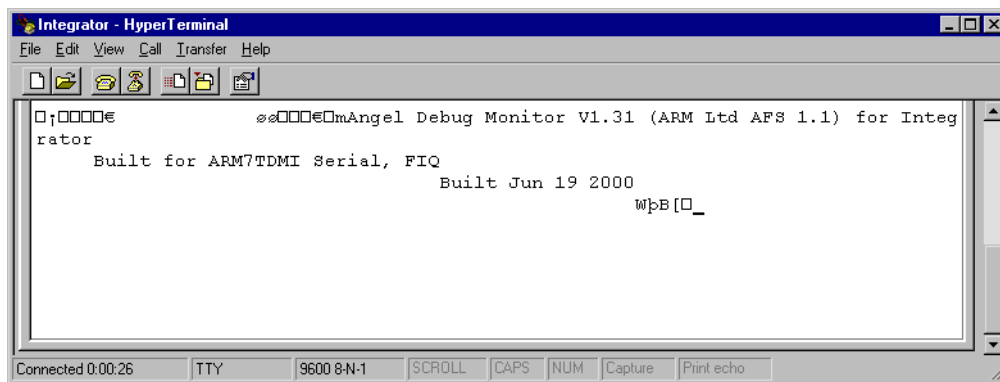


Figure 3-11 Verifying Angel is in flash

5. If you do not see the Angel banner, use boot monitor to identify the images in flash and set the boot image to be the Angel image.
6. If you still do not see the Angel banner, follow the instructions in *Loading an image into flash* on page 3-18 to load the Angel image for your board. The Angel image for the Integrator is located in:

AFSv1_4\images\Integrator\

The Angel project files for the Integrator are located in:

AFSv1_4\Source\angel\Integrator.b\

Note

If you download Angel to the target system, Boot Monitor can load it as image number 0 in any available flash block. If you use AFU, you can load it as any image number (ARM uses image number 911 for Angel in the newer development boards). If you cannot succeed in loading Angel to your target system, contact your supplier.

If you have access to the Multi-ICE interface, you can use it to download an image to flash. Multi-ICE is much faster than the serial port for downloading images.

3.4.3 Loading an image into RAM with Angel

The procedure for loading an image into RAM for a debugger that uses Angel is the same procedure as for a debugger that is using Multi-ICE. The only difference is that the Angel DLL must be used instead of the Multi-ICE DLL. See *Loading an image into RAM with Multi-ICE* on page 3-9 and *Running the flash utility from a debugger* for details on loading an image into RAM.

Note

Disable the heartbeat timeout if the application, led.axf for example, does not allow Angel to periodically communicate with the board.

3.4.4 Running the flash utility from a debugger

Reset the development board and establish communication with a debugger:

1. Set the switches on the target board to boot Angel from flash. See *Verifying Angel is in flash* on page 3-13.
2. Reset the target board.
3. Configure your debugger to use the Angel remote_a DLL for the ADP protocol.

The way you connect to Angel depends on the debugger you are using:

armsd The command line must be of the form:
 armsd -adp -port s=1 -linespeed 38400 image.axf

AXD for ADS

See the *Debuggers Guide* supplied with ADS.

4. The debugger attempts to connect with Angel in the target board (see Figure 3-12).

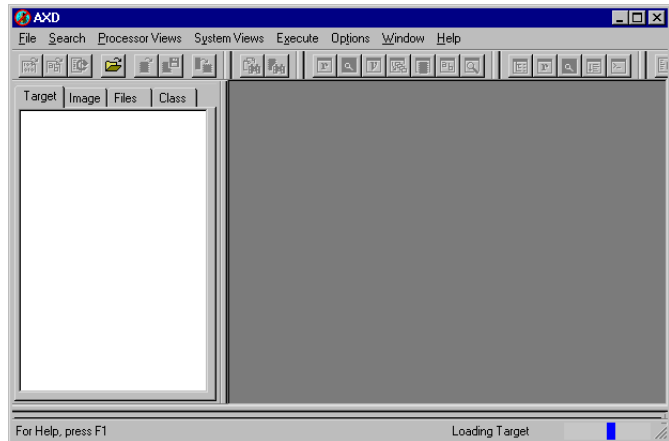


Figure 3-12 Loading Target

5. The debugger connects to the target and the status line displays ADP (see Figure 3-13).

If you do not see ADP, follow the instructions in *Verifying Angel is in flash* on page 3-13.

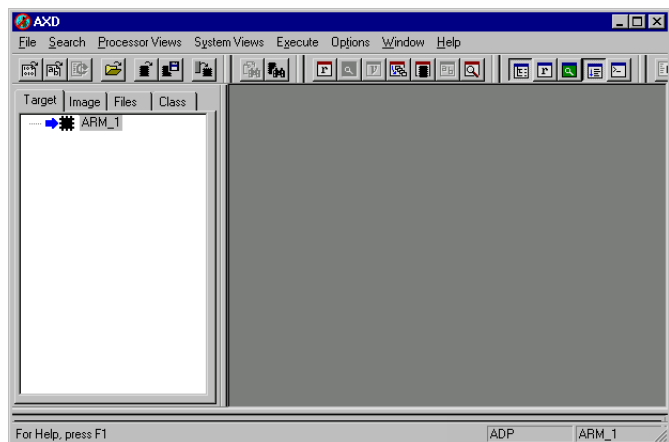


Figure 3-13 Debugger connected to Angel

6. Follow the instructions in *Loading an image into flash with Multi-ICE* on page 3-10 to download a standalone image into flash.

3.4.5 Using a serial port and the boot monitor to load and run images

If you load an image with the boot monitor, it is loaded as image 0. You can use the flash utilities to load an image as a different image number. See *Running the flash utility from a debugger* on page 3-15 or *Loading an image into RAM with Multi-ICE* on page 3-9 for loading applications with Angel and a debugger.

Preparing the board

Follow these steps to prepare your board for loading:

1. Assemble, if necessary, your board and identify the power and data connectors. Refer to the hardware manuals provided with your board.

Note

Do not apply power to the development board yet.

2. Connect and configure a terminal emulator to communicate with boot monitor:
 - a. Locate and install a terminal emulator program that is able to send raw ASCII data files. HyperTerminal is supplied with Windows, but there are also commercial and public-domain emulators available.
 - b. Set up the terminal emulator com port to use 38400 Baud, 8 data bits, no parity, 1 stop bit, and Xon/Xoff (see Figure 3-14).

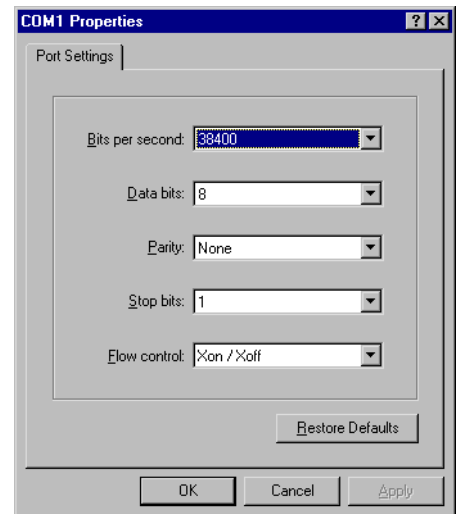


Figure 3-14 Terminal settings

- c. Connect the supplied null-modem cable between the workstation and the first serial port on the development board. (On the Integrator/AP, the first port is the port nearest to the switch box S1.)
- d. Set the configuration switches to use boot monitor. (The settings depend on the board type and version. For the Integrator, set switches S1-1 and S1-4 both to the ON position.)
- e. Power on and reset the board. You should see the boot monitor prompt.
- f. If you do not see the boot monitor prompt, press return on the workstation. If a prompt still does not appear, there is a problem with the terminal emulator software or the hardware.

Loading an image into flash

Follow these steps to load and run an image:

1. If your image is not already in Motorola S-record format, convert it so that it is compatible with the boot monitor. See *Changing the image format* on page 3-6.
2. Set up the terminal emulator as described in *Preparing the board* on page 3-17.
3. Apply power to the development board and establish that you can communicate with the board. You should now see the boot monitor prompt, similar to that shown in Figure 3-15.

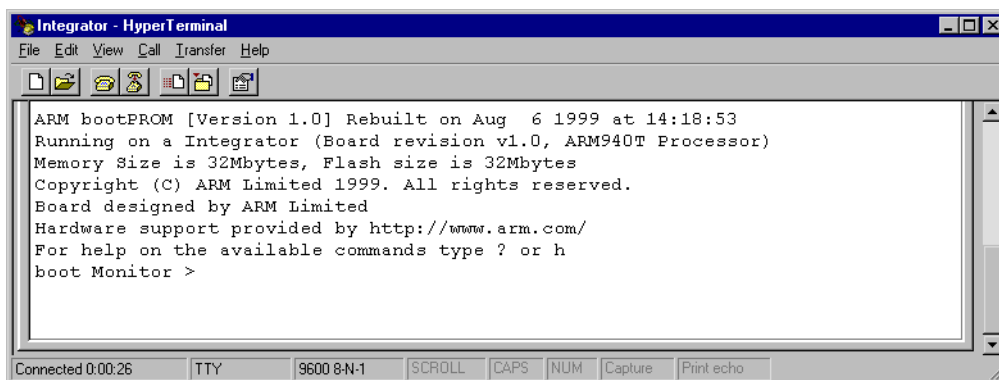


Figure 3-15 Terminal prompt

4. Use the boot monitor to download the image you have built (or one of the prebuilt images from the CD) to the development board. See *Building the LED application* on page 3-4 for build instructions:
 - a. At the command prompt type L to start the Motorola 32 S-record loader.

- b. Select the menu option for transferring an ASCII file (see Figure 3-16 on page 3-19).

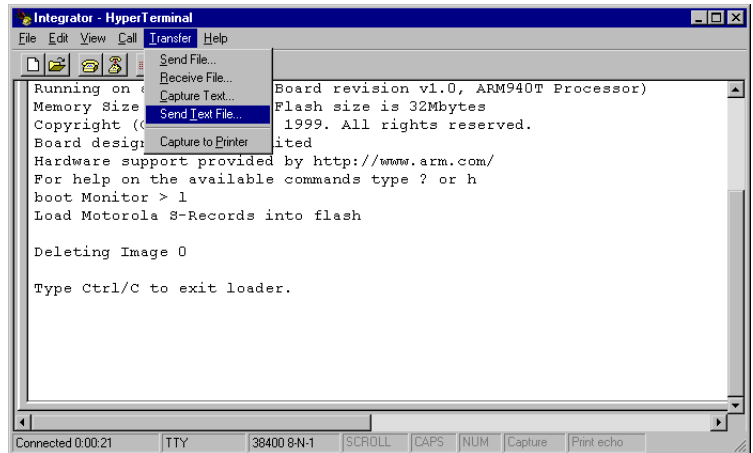


Figure 3-16 ASCII transfer

- c. Select the .m32 file for downloading (see Figure 3-17).

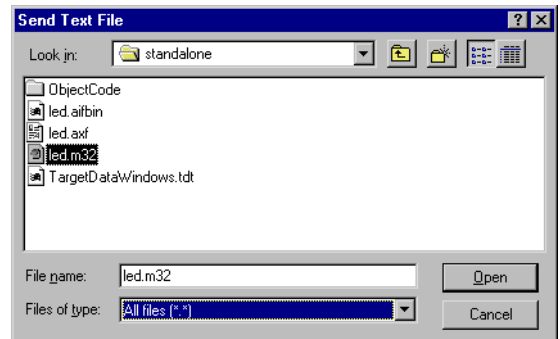


Figure 3-17 Select the file to download

- d. The boot monitor displays a dot for every 64 records loaded. When the terminal emulator has finished sending the file, type Ctrl+C to exit the loader.
 - e. On exit, the loader displays the number of records loaded, the time the load took, and any blocks it has overwritten.
5. Use the BI command in Boot Monitor to set image 0 as the image to run on reset (see Figure 3-18 on page 3-20).

```

ARM bootPROM [Version 1.0] Rebuilt on Aug 6 1999 at 14:18:53
Running on a Integrator (Board revision v1.0, ARM940T Processor)
Memory Size is 32Mbytes, Flash size is 32Mbytes
Copyright (C) ARM Limited 1999. All rights reserved.
Board designed by ARM Limited
Hardware support provided by http://www.arm.com/
For help on the available commands type ? or h
boot Monitor > 1
Load Motorola S-Records into flash

Deleting Image 0

Type Ctrl/C to exit loader.
Downloaded 920 records in 675 seconds.

Overwritten block/s
64

boot Monitor > BI 0
Current Boot Image = 0
New Boot Image     = 0
boot Monitor >

```

Figure 3-18 Setting the boot image

6. Set the switches to boot the selected image from flash. (The switch settings depend on the board type and version. For the Integrator set switches S1-1 to the ON and S1-4 to the OFF position.)

——— **Note** ———

Your development board might use different switch settings. Refer to the documentation supplied with your development board for details on switch settings.

If you require the loaded image to have a different image number, see the description of the ARM Firmware Utilities in the *ARM Firmware Suite Reference Manual*.

7. Reset the development board.
8. The application starts. If you downloaded the LED application, the LEDs on the development board flash.
If the application uses standard output, text is displayed on the terminal emulator or the LCD screen.

3.5 Modifying the application

The settings for the `led.mcp` file produce optimum code size but with reduced debugging capability. You can rebuild the project so that you can easily trace the execution of the application and watch how the μ HAL code is executed.

3.5.1 Changing build options

Follow the steps below to create an application with a better debug view:

1. Start CodeWarrior and open the `led.mcp` project.
2. Select **Edit** → **semihosted settings** to display the settings dialog (see Figure 3-19).

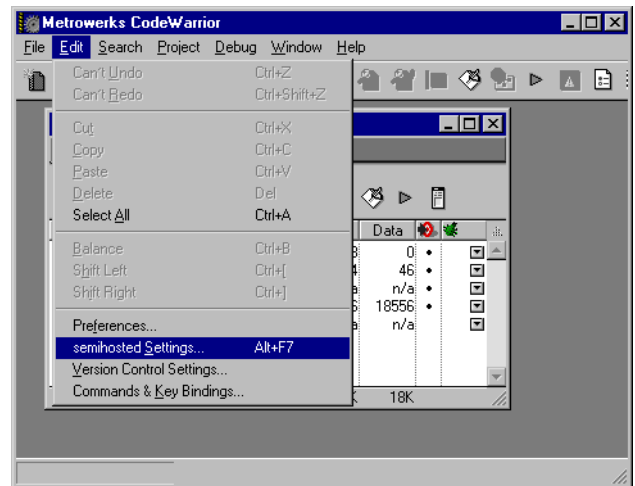


Figure 3-19 Changing the settings

3. Select the **Linker** entry. Check the **Include debugging information** box (see Figure 3-20).

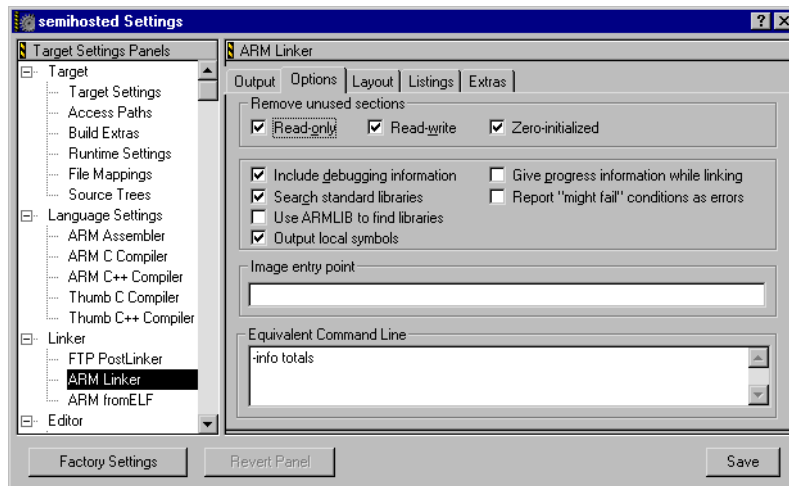


Figure 3-20 Linker options

4. Select the **ARM C Compiler** entry and the **Debug** tab. Change the settings to produce optimum debug view (see Figure 3-20).

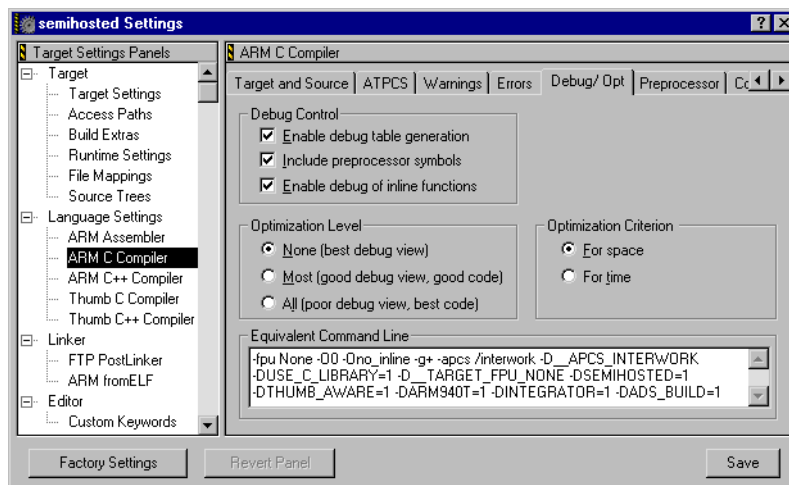


Figure 3-21 Debug options

5. Save the changes.
6. Open the uHAL subproject and change its settings as in step 3.
7. Select **Project** → **Make** to rebuild the uHAL library.

Note

Remember to change the settings back to optimum code size if you rebuild the project later and do not want the full debug information.

3.5.2 Stepping through the code

Follow the steps below to step through the code and see the call to the μ HAL library code:

1. Start AXD and connect to the target board. This requires either Angel on the target board or the Multi-ICE hardware. See *Starting the debugger* on page 3-8.
2. Load the semihosted led.axf image.
3. Select **Processor views** → **Source** and load the file AFSv1_4\Source\uHAL\Sources\led.c (Figure 3-22).

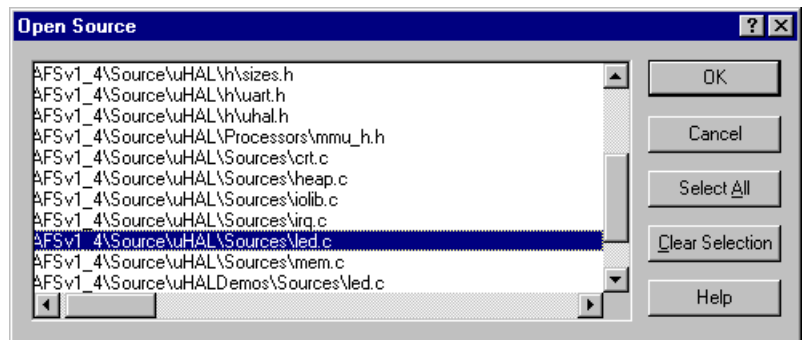
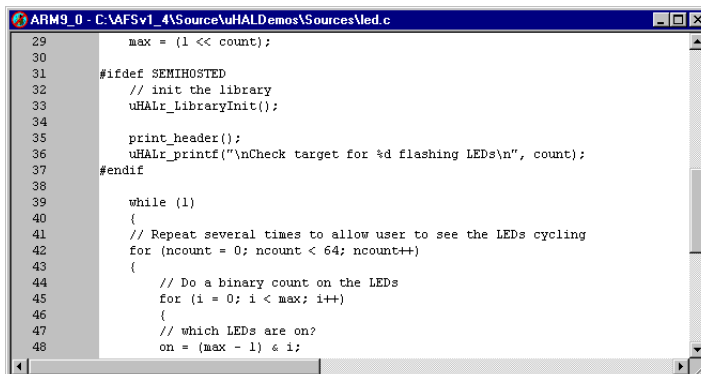


Figure 3-22 Loading the source files

4. Set a breakpoint in led.c to stop when the uHALr_WriteLED() call is reached (see Figure 3-23).



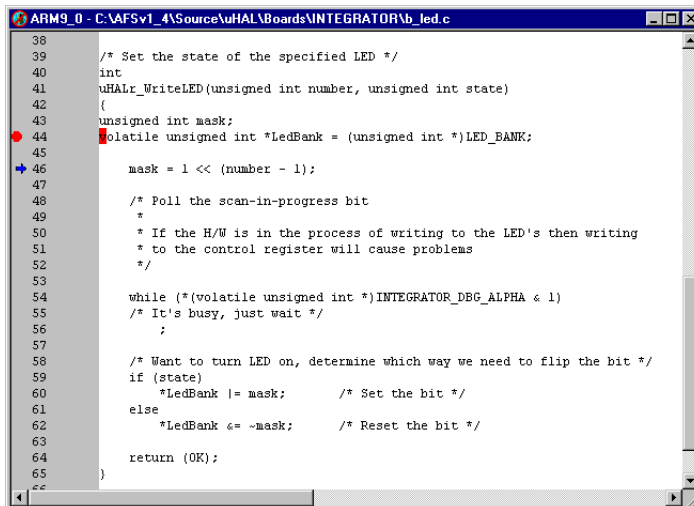
```

29     max = (1 << count);
30
31     #ifdef SEMIHOSTED
32         // init the library
33         uHALr_LibraryInit();
34
35         print_header();
36         uHALr_printf("\nCheck target for %d flashing LEDs\n", count);
37     #endif
38
39     while (1)
40     {
41         // Repeat several times to allow user to see the LEDs cycling
42         for (ncount = 0; ncount < 64; ncount++)
43         {
44             // Do a binary count on the LEDs
45             for (i = 0; i < max; i++)
46             {
47                 // which LEDs are on?
48                 on = (max - 1) & i;

```

Figure 3-23 Breakpoint in led.c

5. Select **Execute** → **Go** to start the application.
6. After the debugger stops on the breakpoint, select **Execute** → **Step in** to continue execution.
7. The debugger shows the code execution continuing in the b_led.c file. This file contains the uHAL_WriteLED() code that is specific to the Integrator board (see Figure 3-24).



```

38
39  /* Set the state of the specified LED */
40  int
41  uHALr_WriteLED(unsigned int number, unsigned int state)
42  {
43      unsigned int mask;
44      volatile unsigned int *LedBank = (unsigned int *)LED_BANK;
45
46      mask = 1 << (number - 1);
47
48      /* Poll the scan-in-progress bit
49      *
50      * If the H/W is in the process of writing to the LED's then writing
51      * to the control register will cause problems
52      */
53
54      while (*(volatile unsigned int *)INTEGRATOR_DBG_ALPHA & 1)
55          /* It's busy, just wait */
56          ;
57
58      /* Want to turn LED on, determine which way we need to flip the bit */
59      if (state)
60          *LedBank |= mask;      /* Set the bit */
61      else
62          *LedBank &= ~mask;     /* Reset the bit */
63
64      return (OK);
65  }

```

Figure 3-24 The µHAL library code for writing to an LED

Chapter 4

Running the Timer Application

This chapter describes how the system-timer application uses μ HAL. It contains the following sections:

- *How the application uses μ HAL* on page 4-2
- *Running the application* on page 4-5.

4.1 How the application uses μ HAL

The system-timer application installs a timer interrupt to update a variable. A loop in `main()` contains the code that reads the variable and outputs its value to the standard output port. The examples in this section list code from the timer application. To view the code, either:

- Open the `AFSv1_4\Source\uHALDemos\Sources\system-timer.c` file in a text editor.
- Use the CodeWarrior IDE to open the `system-timer.mcp` file for your board. For example, to use the project file for an Integrator board with an ARM940T processor, use the `integrator940T.b` project file in the directory `AFSv1_4\Source\uHALDemos\Build\` and double-click on the entry `system-timer.c`.

4.1.1 The interrupt function

The function in Example 4-1 is installed as the interrupt simply increments the value of a global variable. This function is passed to a μ HAL interrupt installation function.

Functions that are installed as interrupt handlers by μ HAL must have a single integer parameter and a void return type.

Example 4-1 The interrupt function

```
// called by IRQ Trap Handler when the timer interrupts
static int OSTick = 0;
void TickTimer(unsigned int irq)
{
    OSTick++;
}
```

4.1.2 Displaying the board details

Before the interrupt function is installed, the board details are loaded into an `infoType` structure and displayed as shown in Example 4-2 on page 4-3.

The `uHALr_GetPlatformInfo()` function source is in the `board.c` file. There is an implementation of this routine for each board that returns the board-specific information.

Example 4-2 Display the board details

```

int main(int argc, int *argv[])
    int i, j;
    infoType platformInfo;

    print_header();

    // who are we?
    uHALr_GetPlatformInfo(&platformInfo);
    uHALr_printf("platform Id :0x%08X\n",platformInfo.platformId);
    uHALr_printf("memory Size :0x%08X\n", platformInfo.memSize);
    uHALr_printf("cpu ID      :0x%08X\n", platformInfo.cpuId);

```

4.1.3 Installing the timer interrupt

Example 4-3 on page 4-4 shows how the timer is installed. There are several steps to complete before the timer interrupt is functioning:

1. `uHALr_InitInterrupts()` is called once on startup to initialize the μ HAL internal interrupt structures. This must be called before installing a new IRQ handler with μ HAL.
2. `uHALr_InitTimers()` must be called before any other timer function. This function:
 - Sets the host timer to `T_LOCKED`, if the application is semihosted, so that the host timer cannot be modified by the application.
 - Initializes the μ HAL internal timer structures.
 - Resets all timers to a known state (it sets the internal delays to a predefined value and sets all timers off).
3. The call to `uHALr_RequestSystemTimer()`:
 - installs a handler for the system timer
 - sets up the internal structures
 - stops, and does not restart, the timer.

By default, the system timer is set to tick once every millisecond.
4. `uHALr_InstallSystemTimer()` starts the timer and enables the interrupt associated with it.

Example 4-3 Install the timer

```
// Install new trap handlers and soft vectors
uHALr_InitInterrupts();
// initialise the timers
uHALr_InitTimers();

// initialise the tick count
OSTick = 0;
uHALr_printf("Timer init\n");
if (uHALr_RequestSystemTimer(TickTimer,
    (const unsigned char*)"test") <= 0)
    uHALr_printf("Timer/IRQ busy\n");

// Start system timer & enable the interrupt.
uHALr_InstallSystemTimer();
```

4.1.4 Displaying the timer value

Example 4-4 shows the loop that continuously flashes the LED and prints the timer value. The μ HAL LED functions are used to turn the LED on and off.

Example 4-4 Display the timer count

```
// loop flashing a led and giving out the tick count
for (j = 0;; j++)
{
    if (j & 1)
        uHALr_SetLED(1);
    else
        uHALr_ResetLED(1);

    uHALr_printf("Tick count is 0x%05X\n", OSTick);

    // Wait around for a bit..
    for (i = 0; i < 1000000; i++)
        ;
}
print_end();
return (OK);
}
```

4.2 Running the application

To build and run the timer application, follow the steps below:

1. Build an executable image for the application from the makefile or the CodeWarrior project file located in the directory for your development board. For example, to use the project file for an Integrator board with an ARM940T processor, use
AFSv1_4\Source\uHALDemos\Build\Integrator940T.b\simple-timer.mcp.
For an Integrator board with a generic ARM7TDMI, use
AFSv1_4\Source\uHALDemos\Build\Integrator.b\simple-timer.mcp.
2. Follow the instructions in *Loading an image into RAM with Multi-ICE* on page 3-9 to download the semihosted application to the development board.
3. Use AXD to start the application. The timer values are displayed in the console as shown in Figure 4-1.

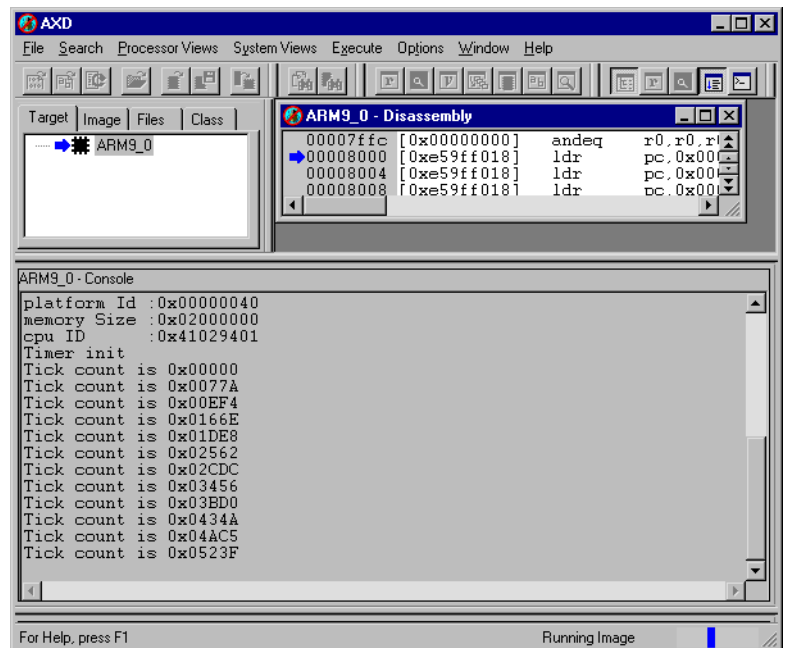


Figure 4-1 Timer application output

Chapter 5

Creating a New Application

This chapter describes how to create and build a new application. It contains the following sections:

- *How to create a new application* on page 5-2
- *Extending the existing directory structure* on page 5-3
- *Creating a separate directory structure* on page 5-10.

5.1 How to create a new application

There are several ways to create a new application:

- Modify one of the existing demonstration applications. This method does not require detailed knowledge of how applications are built, but it is only suitable for simple applications. See *Modifying the application* on page 3-21 for details on using the CodeWarrior IDE and an existing application.
- Create new directories within the existing directory structure and place your source code there. You must also create either makefiles or project files that control how your application is built. Your make or project files must use the correct relative path to access the library and include files that match your target hardware.

This method can be used for medium or large applications. Reusing the existing directory structure simplifies selecting the correct library and include files.

- Create a new directory structure and copy the necessary library and include files from the AFS installation directories. In addition to creating the make or project files that use files in the new paths, you must also ensure that you have copied the correct build variants of the files for your target hardware.

This method gives the most flexibility, but it requires more understanding of how applications are constructed. If you only build for one processor and development board combination, you can use this method to simplify the directory structure.

5.2 Extending the existing directory structure

This section describes how to extend the standard AFS directory structure to provide a working area for your own applications. This method is simple and fast, but it has the disadvantage that common library files are shared between different applications. This might result in someone changing a library build without your knowledge.

If you want more control of the build process and included files, create a separate directory structure and copy the required library files to it (see *Creating a separate directory structure* on page 5-10).

5.2.1 Using CodeWarrior projects

Follow the steps below to create new directories for CodeWarrior projects:

1. Create new directories for your project and source files:
 - a. Change directory to the Source directory in your AFS installation directory (for example, AFSv1_4\Source).
 - b. Create a new directory to hold your work. For example, AFSv1_4\Source\User.
 - c. Create a Sources subdirectory in your new directory.
 - d. Create a Build subdirectory in your new directory.
 - e. Create a directory within Build that matches your development board and target processor.
For example, create a directory called Integrator940T.b if you have an Integrator board with an ARM940T processor. Create a directory called Integrator.b if you have an Integrator board with a generic ARM7TDMI processor.
 - f. Create new source files in your Sources directory to add to the project. For example, copy AFSv1_4\Source\uHALDemos\Sources\led.c and AFSv1_4\Source\Source\pr_head.c to AFSv1_4\Source\User\Sources\led2.c and AFSv1_4\Source\User\Sources\pr_head.c.
2. Modify an existing project file to use different source files:
 - a. Copy a project file from one of the existing demonstration directories to the subdirectory for your hardware.
For example, copy AFSv1_4\Source\uHALDemos\Build\Integrator940T.b\led.mcp to AFSv1_4\Source\User\Build\Integrator940T.b\led2.mcp.
 - b. Open the project file in the new destination directory.

- c. Delete the existing source files from the project by selecting the files and pressing **Delete** (see Figure 5-1).

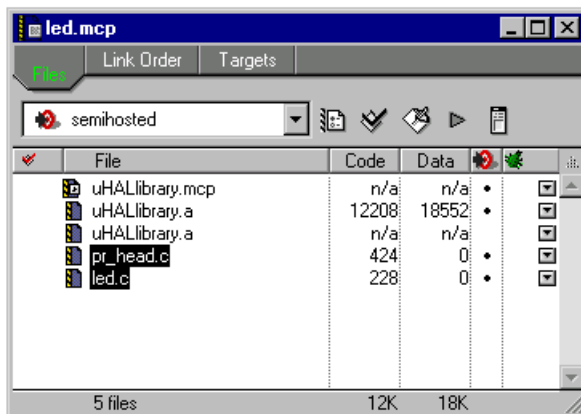


Figure 5-1 Delete existing source files from project

- d. Confirm the delete by selecting **OK** (see Figure 5-2).

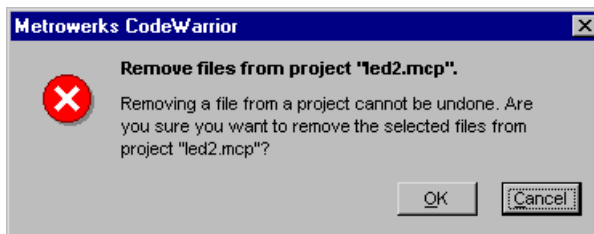


Figure 5-2 Confirm delete

- e. Add your source files to the CodeWarrior project.
For example, Figure 5-3 shows adding
AFSv1_4\Source\User\Sources\led2.c and
AFSv1_4\Source\User\Sources\pr_head.c

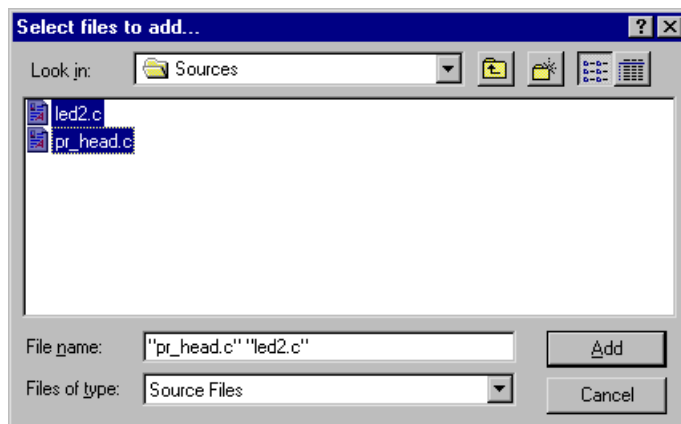


Figure 5-3 Adding new files to the project

- f. Add your source files to both the semihosted and standalone builds (see Figure 5-4).

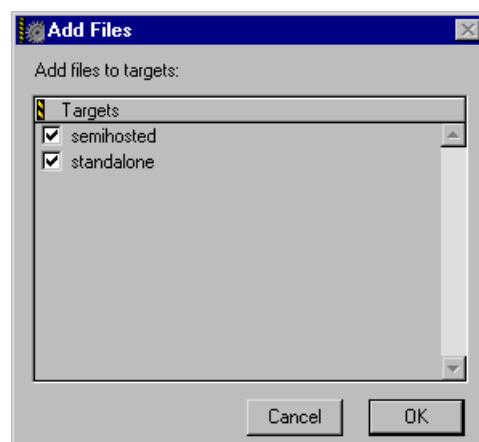


Figure 5-4 Targets for source files

- g. Verify that the correct library build targets match the overall build targets by selecting the **Targets** tab (see Figure 5-5).

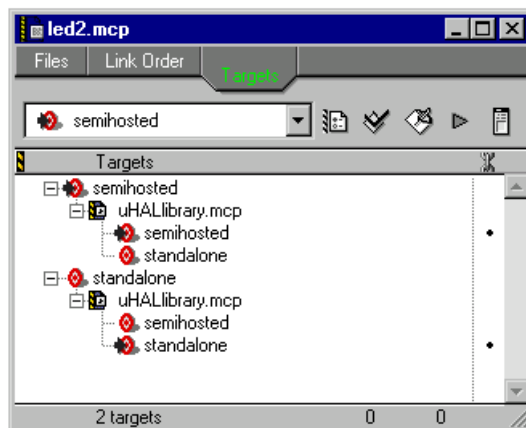


Figure 5-5 Targets for library

3. The original `led.mcp` project used the basic uHAL library, but you might require the PCI or flash libraries for your new application if you are calling PCI or flash routines. If so, add the prebuilt library file to the project.

For the flash library and Integrator, select the file from the directory containing builds for all Integrator boards in `AFSv1_4\lib\Integrator`. For the PCI library and Integrator, you can select one of:

- the generic library file from `AFSv1_4\lib\Integrator`
- a generic Thumb-aware file from `AFSv1_4\lib\IntegratorT`
- a library which is built for the specific processor.

For example, the files in `AFSv1_4\lib\Integrator940T` are built for the Integrator board with an ARM940T processor. The generic libraries are build for an ARM7 processor running in ARM mode. The files in the `IntegratorT` directory are build for an ARM7TDMI with Thumb interworking.

Select the library variant that matches the semihosted or standalone build options. `PCI_u_.a` is the semihosted PCI library. `PCI_ur.a` is the library for applications in ROM. See the *ARM Firmware Suite Reference Guide* for more details on library naming. See Figure 5-6.

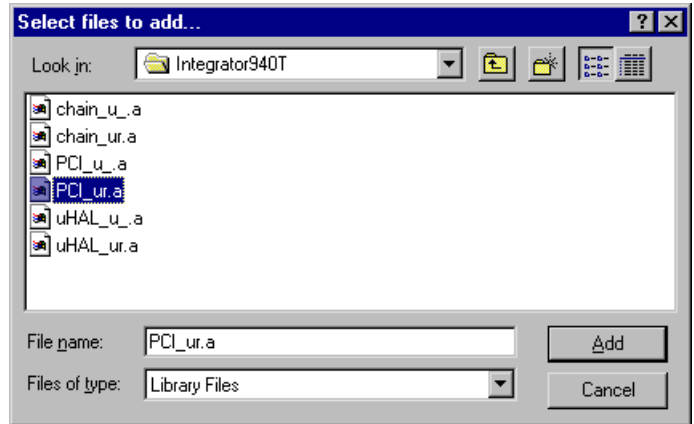


Figure 5-6 Adding a prebuilt library to the project

4. Select **Project** → **Make** to build your new application.

Because uHALlibrary is a subproject in your application, the library is automatically rebuilt if necessary.

If you included library archive files directly in the application, for example, PCI_ur.a, the library is not rebuilt and you cannot change its build options by changing project options. If you require different build options for a library, you must select a different library variant.

5.2.2 Using makefiles

You can use makefiles to control your build process instead of using project files. The existing makefile and .in include files from the uHALDemos directory can be used as a starting point.

To extend the existing directory structure and customize your makefiles:

1. Create new directories in your AFS installation directories as described in step 1 of *Using CodeWarrior projects* on page 5-3.
2. Create two output directories in the build directory for your hardware called semihosted and standalone. For example, the full paths are AFSv1_4\Source\User\Build\Integrator940T\semihosted and AFSv1_4\Source\User\Build\Integrator940T\standalone.
3. Create an include file to define your build environment to the make process:
 - a. Copy environ.in from AFSv1_4\Source\uHALDemos\ to AFSv1_4\Source\User\.

- b. Open `environ.in` in a text editor and modify the list of boards. Enter the name of your hardware subdirectory without the trailing `.b`.
 # The set of boards/subdirectories that we need to build.
`BOARDS = Integrator940T`
 - c. Save the modified file.
4. Create an include file that specifies which applications you are building:
 - a. Copy `common.make` from `AFSv1_4\Source\uHALDemos\Build` to `AFSv1_4\Source\User\Build`.
 - b. Open `common.make` in a text editor and modify the list of target applications. Enter the name of your application and the file type:
`$(TARGET)_all: $(TARGET)/led2.axf \
 $(TARGET)/led2.bin`
 - c. Modify the build instructions for the `led` application to use the new application name and source files as shown in Example 5-1.

Example 5-1 makefile

```

*****
# LED flashing program
*****
$(TARGET)/led2.o : $(SOURCES)/led2.c $(SYSTEM_INCLUDE_FILES)
$(ARMCC) $(CFL) $(CFLAGS) $(CDEFS) -c $(INCL) $< -o $*.o

ifeq ($(TARGET),semihosted)
LEDOBJS=$(TARGET)/led2.o $(TARGET)/pr_head.o
else
LEDOBJS=$(TARGET)/led2.o
endif

$(TARGET)/led2.axf: $(LEDOBJS) $(LIBRARIES) $(UHALLIB)
$(ARMLINK) $(LFLAGS) $(LDEFS) $(SYM_LIST) \  

    $(TARGET)/led2.sym $(LIBRARIES) $(UHALLIB) $(LEDOBJS) \  

    -o $(TARGET)/led2.axf

$(TARGET)/led2.bin: $(TARGET)/led2.axf
$(FROMELF) $(ELFDEFS) $(TARGET)/led2.axf -bin $(ELF_OUTPUT) \  

$(TARGET)/led2.bin

```

- d. Delete the other demo application target instructions (for `bubble`, `exception`, `file-io`, `hello`, and so on) from the include file. Do not, however, delete the library or cleanup instructions.
 - e. Save the modified file.

5. Create a makefile that controls the overall build process by copying makefile from AFSv1_4\Source\uHALDemos\Build\Integrator940T.b to AFSv1_4\Source\User\Build\Integrator940T.b

The makefile uses the line:

```
include $(UHAL_BASE)/Build/$(BOARD_NAME).b/board.in
```

to select the settings for your development board and processor.

Note

Because the BOARD_NAME variable is used to retrieve files from other directories, the board names in your User\Build directory must be the same as the ones in the uHAL\Build directory.

6. Change location to the board directory and use the make command to build your application.

Note

Using make requires additional software not provided with ADS or AFS. If you are using Windows, you can download the public-domain Cygnus command shell.

5.3 Creating a separate directory structure

This section describes how to create a new independent directory structure to hold both your source and library files. This method gives you more control over how your applications are built, but it requires more knowledge of the internal stages of the build process.

Caution

You must copy the correct library variants for your development board and processor. Use the directory names to identify the board or processor settings used in the library.

5.3.1 Using the CodeWarrior IDE

If you are using the CodeWarrior IDE, the search paths for the project must match the board and processor. This section describes how to modify the access paths and preprocessor settings for a project. See the CodeWarrior IDE documentation for details on creating and modifying projects:

1. Create new directories and project files. See *Extending the existing directory structure* on page 5-3 for examples of how to use existing CodeWarrior project files as a starting point for new projects.
2. Change the access paths that CodeWarrior uses to locate user files:
 - a. Select **Edit** → **Project Settings** to display the settings dialog.
 - b. Select **Access Paths** from the left panel (see Figure 5-7).

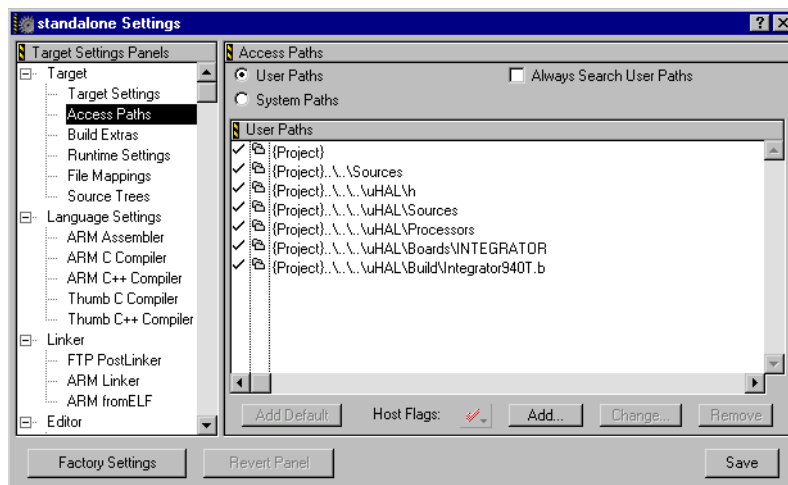


Figure 5-7 Access paths

- c. Click on one of the user paths to select it.
- d. Click **Change** to display the dialog for selecting a new path. Select the new path and click **OK**. See Figure 5-8.

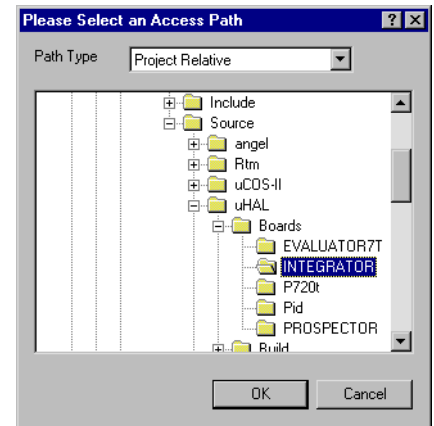


Figure 5-8 Relative path

- e. Use **Remove** to delete a selected path that is not used in your new directory structure.
- f. Use **Add** to add a new path. If you are adding an absolute path, use the **Path Type** drop-down list and select absolute path (see Figure 5-9).

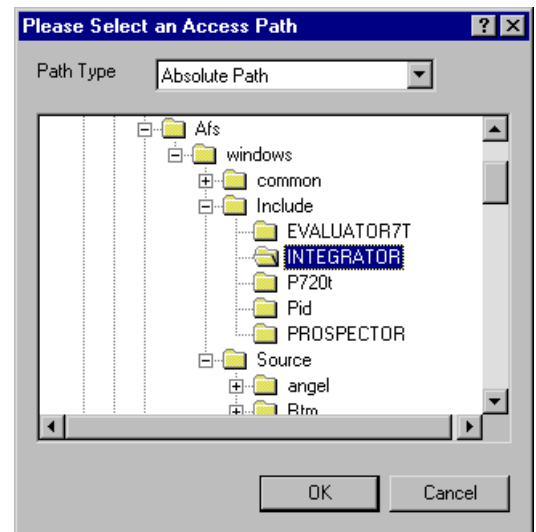


Figure 5-9 Absolute path

3. Change the predefineds and preprocessor settings for the assembler and compilers:
 - a. Set the project defines to match your build target. Select the **Preprocessor** tab to set defines for the compilers (see Figure 5-10).

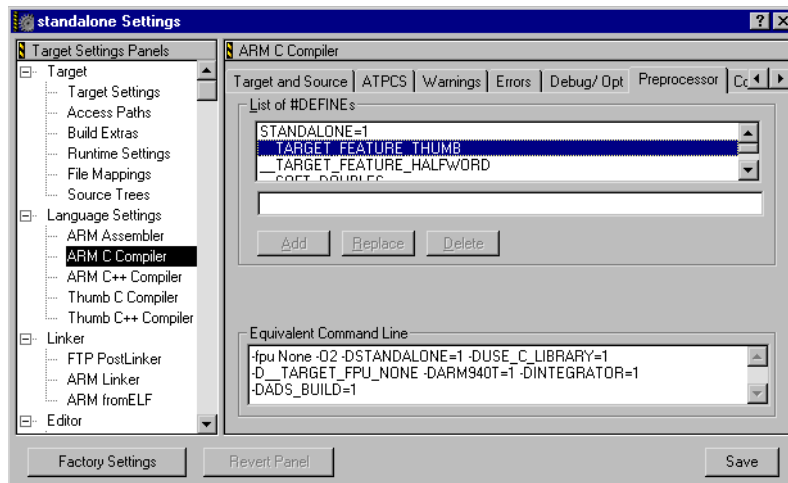


Figure 5-10 Compiler preprocessor settings

- b. Select the **Predefines** tab to set defines for the assembler (see Figure 5-11).

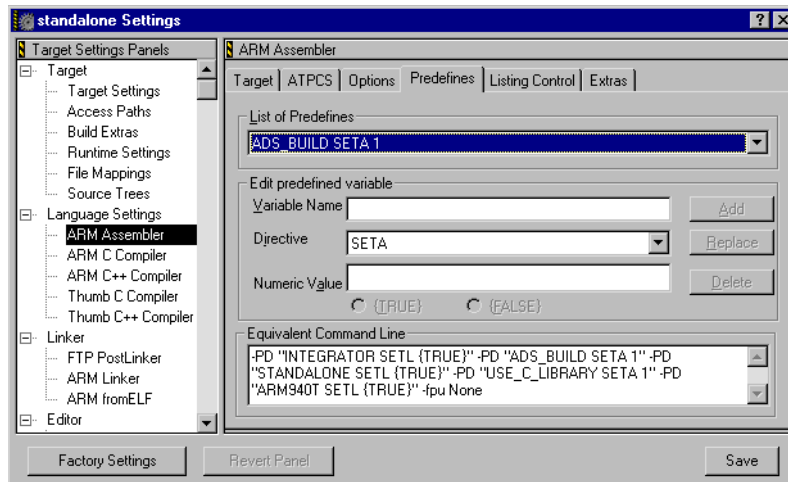


Figure 5-11 Assembler predefineds

5.3.2 Using makefiles

You must be familiar with the build process, the library files, and makefiles to control your build process. Instructions on the make process are beyond the scope of this document. This section, however, covers some of the key concepts and issues.

Create your new directory structure and copy the necessary library archives and header files into a directory within that structure. If you are only building for one board and processor combination, you can copy just the files needed for that build target. If you are building for different board or processor combinations, you must copy all the files that are used for your build targets and use directory naming to identify which files are used for each target build.

Create a makefile that matches your new directory structure, library archives, and build target. If you use the existing make and include files as a starting point, edit the defines that set the library base paths. Extensive use is made of path-relative directory names. The text in Example 5-2 shows how the `rules.in` file calculates paths for include files:

Example 5-2 Relative path names in `rules.in`

```
...

# What is the set of include files for this board?
ifndef UHAL_BOARD_INCLUDES
    UHAL_BOARD_INCLUDES= $(UHALBOARDDIR)/platform.h \
        $(UHALBOARDDIR)/platform.s $(UHALBOARDDIR)/target.s
endif

...

# *****
# Where are the sources?
# *****
# where stuff is (in uHAL)
# [A] The directories
UHALINCLUDEDIR = $(UHAL_BASE)/h
UHALBOARDDIR   = $(UHAL_BASE)/Boards/$(BOARD_TYPE)
UHALPROCESSORDIR = $(UHAL_BASE)/Processors
UHALARMDIR     = $(UHAL_BASE)/Processors/$(PROCESSOR_NAME)
UHALLIBDIR     = $(UHAL_BASE)/Sources/
UHALTOOLDIR    = $(UHAL_BASE)/tools/

# [B] Sources
UHALLIBSOURCES= $(UHALLIBDIR)/boot.s $(UHALLIBDIR)/led.c \
    $(UHALLIBDIR)/iolib.c $(UHALLIBDIR)/timer.c \
    $(UHALLIBDIR)/heap.c $(UHALLIBDIR)/irq.c \
```

```

$(UHALLIBDIR)/irqlib.s $(UHALLIBDIR)/irqtrap.s \
$(UHALLIBDIR)/switrap.s $(UHALLIBDIR)/undeftrap.s \
$(UHALLIBDIR)/crt.c $(UHALLIBDIR)/mort.s \
$(UHALLIBDIR)/external.s $(UHALLIBDIR)/cpumode.s \
$(UHALLIBDIR)/divide.s $(UHALLIBDIR)/support.s
UHALPROCESSORSOURCES= $(UHALPROCESSORDIR)/cache.c \
$(UHALPROCESSORDIR)/control.s $(UHALPROCESSORDIR)/mmu.s

# [C] Include files
UHALLIBINCLUDES = $(UHALINCLUDEDIR)/uhal.h \
$(UHALINCLUDEDIR)/cdefs.h $(UHALINCLUDEDIR)/bits.h \
$(UHALINCLUDEDIR)/sizes.h
UHALPROCESSORINCLUDES = $(UHALPROCESSORDIR)/mmu_h.h \
$(UHALPROCESSORDIR)/mmu_h.s \
$(UHALPROCESSORDIR)/mmumacro.s

```

The build process is considerably simplified if you use prebuilt libraries rather than including the instructions for building a library variant. If you use prebuilt libraries however, you must ensure that you are using the correct variant for your build options.

Note

If you are using PCI or flash functions in your application, your makefile must include a path to a prebuilt PCI or flash library. Prebuilt .a library archives are located in the subdirectory in AFSv1_4\lib\ that corresponds to your development board and processor.

Chapter 6

Troubleshooting and Frequently Asked Questions

This chapter describes solutions to problems that occur when producing an application, and provides answers to general questions about AFS. It contains the following sections:

- *Frequently asked questions* on page 6-2
- *Troubleshooting* on page 6-5.

6.1 Frequently asked questions

This section gives the answers to some frequently asked questions about the AFS.

6.1.1 Does AFS support Thumb?

The CodeWarrior IDE project files installed for the AFS components and demo applications use a define in the **ARM C Compiler** and **ARM Assembler Preprocessor** tabs of the target settings window to select Thumb support:

```
-DTHUMB_AWARE=1
```

There are different directories for Thumb and non-Thumb builds:

Integrator.b This directory contains project files that build non-Thumb versions. These builds can be run on any supported processor.

IntegratorT.b

This directory contains project files that build Thumb versions. These builds can be run on any supported processor that can execute Thumb code.

To build applications with the makefile, set the build variable:

```
make THUMB_AWARE=1
```

6.1.2 How do I build AFS components using ADS?

There are ADS CodeWarrior IDE project files installed for the AFS components and demo applications.

The makefile builds for ADS by default. The build is controlled by the following build variable:

```
make ADS_BUILD=1
```

6.1.3 How do I use the C library?

The CodeWarrior IDE project files installed for the AFS components and demo applications use a define in the **Processors** tab of the target settings window to select the ADS C library:

```
-DUSE_C_LIBRARY=1
```

The CodeWarrior IDE projects that build applications with the ADS C library require an additional assembler predefine:

```
USE_C_LIBRARY SETA TRUE
```

and an additional C preprocessor #define:

```
USE_C_LIBRARY=1
```

or use `-DUSE_C_LIBRARY=1` on the compiler command line.

UNIX makefiles that build using the ADS C library must define the additional makefile build variable:

```
make USE_C_LIBRARY=1
```

The main effect of building with the ADS C library is that μ HAL now defines the heap and stack base and size and exports these to be used by the C library memory management routines. Program start-up and termination are still controlled by C library routines, but additional routines in the C library are used to perform some of the initialization.

6.1.4 Is μ HAL free?

The μ HAL demonstration and example programs are freely reusable and redistributable so long as they are used on ARM-based platforms. The other ARM Firmware Suite components must be licensed from ARM.

6.1.5 Can I use μ HAL in my project?

You can use μ HAL in your commercial projects. Refer to your license agreement for details, or contact your supplier if you require further information.

————— Note —————

Because there are portions of the firmware base level that are not free, you must be careful about the parts that are used and where they are used. Contact your ARM sales representative for more information.

6.1.6 What boards are supported?

The current set of supported evaluation boards consist of:

- Integrator
- Prospector
- ARM Evaluation Board (Evaluator 7T)
- Intel IQ80310.
- Intel IQ80321.
- Agilent AAED-2000.

6.1.7 How do I use boot monitor with Multi-ICE?

Boot monitor does not use the Multi-ICE interface, but you can load and debug applications using Multi-ICE and the flash utilities:

1. Install the Multi-ICE server software on your PC.
2. Configure your debugger to use the Multi-ICE interface.
3. Connect the Multi-ICE cable to the JTAG connector on the processor card.
4. Power-on the development board.
5. Use the **Load image** command from the debugger to load and debug an image in the RAM of the development board, or load afu and use the debugger console to store images in flash.

Once a standalone image has been loaded into flash with Multi-ICE, you can use the boot monitor and a serial port to select the image to run on reset.

6.1.8 How can I verify that Angel is installed?

To test whether Angel is installed on your board, refer to the instructions in *Verifying Angel is in flash* on page 3-13.

6.2 Troubleshooting

The topics below list solutions to problems that might occur when you build an application using μ HAL.

6.2.1 μ HAL does not work with my processor

μ HAL supports the following processors:

- ARM7TDMI (in the Integrator and Integrator.b directories)
- ARM720T
- ARM740T
- ARM920T
- ARM940T
- ARM946E-S (preliminary support only)
- ARM966E-S
- StrongARM (SA110 and SA1100).
- XScale
- ARM 10 (preliminary support only).

If the processor you want to use is not on this list, you might still be able to use all or part of the μ HAL source code or definitions in your application. The generic code is based on an ARM7TDMI processor.

6.2.2 The boot switcher fails to run an image from flash

Take one of the following actions to remedy this condition:

- Check that the boot image number is correct by using the boot monitor BI command.
- Check that the image is correctly programmed by using the boot monitor V command.
- For the Integrator board, use the boot monitor DC command in the extended command mode to check that the clock settings in the SIB are reasonable values for the core module you are using.
- Check the switch settings for your board.

6.2.3 AXD cannot download to Integrator flash memory

The following actions may solve the problem:

1. Download to flash with switch 1 in the on position.

- 2. Power cycle the board.
- 3. Start AXD and download to flash.
- 4. Put switch 1 down again.
- 5. Power cycle the board to run the image.

If you still have problems, clear the flash with HyperTerm (using the e command in the boot monitor) and start again with switch 1 on.

See *The switch settings do not behave as expected* or the documentation supplied with your development board for more information on switch settings.

6.2.4 The switch settings do not behave as expected

The function of the switches depends on the development board and the firmware version. The function of the switch settings for the Integrator board are described in Table 6-1. Refer to the documentation supplied with your development board for more information on switch settings. For example, on the Prospector board, switch U25-5 selects the boot monitor

Table 6-1 Integrator switch settings

Switch 1 - 4 settings	Result
ON OFF OFF ON	The boot monitor is run on reset.
OFF OFF OFF ON	The application image at address 0x24000000 is run on reset. This is the default address for images loaded with the boot monitor L command.
ON OFF OFF OFF	The image designated as the boot image is run on reset. Use the boot monitor BI command to select the image number.

6.2.5 Integrator images fail to load after Multi-ICE Auto-Configuration

Take one of the following actions to remedy this condition:

- Reset the system.
- Power the system OFF and then ON again.
- Set the REMAP bit (bit 2 in the CM_CTRL register at 0x1000000C) from the debugger.

6.2.6 Exception vector errors when using Multi-ICE

The message Unable to set breakpoints on exception vectors is displayed when using Multi-ICE on Integrator.

- Depending on the value of \$vector_catch, Multi-ICE attempts to write to the vectors. This error occurs if the memory has not been remapped. Refer to the *Multi-ICE User Guide*.

6.2.7 I cannot enable a timer that has not been requested

You must allocate a timer by requesting it before you can use it in any way, including enabling it. The System Timer ID is available using uHALIr_GetSystemTimer().

6.2.8 Enabled timer interval is too long

The interval count is reloaded when the timer is enabled, so it is the full interval duration before the timer event occurs again. Load the correct timer interval before you enable the timer.

6.2.9 Terminal emulator does not work with boot monitor

The line parameters for the boot monitor are the defaults for the μ HAL port for that board. Look in the porting documentation for your board for details. For many ARM boards, the defaults are 38400 baud, 8 data bits, no parity, and one stop bit.

Either TTY or VT100 emulation should work. You must enable Xon/Xoff flow control for the emulator. Where there are two ports, look in the board documentation or in the source to identify the port to use (or more simply, try both).

6.2.10 I cannot use ELF format in my application

The ADS utility fromELF is provided to generate other file formats from an ELF image. See your ADS documentation for details on using fromELF.

6.2.11 Demo applications run slower as standalone

The μ HAL Board Demo routines appear to show differences in running time between the semihosted and standalone variants. The semihosted variants can run approximately four times faster when not using the processor cache.

This difference is due to the semihosted code being run from system RAM (linked to execute at location 0x8000 which is the default location for debugger execution).

The standalone code is linked to run directly from the flash memory in which it is stored. Flash memory accesses are slower than RAM accesses giving the performance loss.

The system test and sieve demo applications show how the use of system caches improve routines performance from slower memory.

The standalone code can be linked to run from RAM for direct performance comparison. You can achieve this by setting the Read Only address for the linker to 0x8000.

When programming an image linked for RAM into flash memory, use the AFU to program an image into the flash (the location in flash is not important). The flash image contains all the information in the footer to ensure that the code can be copied to the correct location and executed. An example of the code required to use this information is in the ARM boot monitor.

Glossary

	ADP	Angel Debug Protocol, see <i>Angel</i>
	ADS	See <i>ARM Developer Suite</i> .
	ADU	See <i>ARM Debugger for UNIX</i> .
	ADW	See <i>ARM Debugger for Windows</i> .
	AFU	See <i>ARM Flash Utility</i> .
	AFS	See <i>ARM Firmware Suite</i> .
	Angel	Angel is a program that enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either ARM state or Thumb state.
	ANSI	American National Standards Institute.
	API	See <i>Application Programming Interface</i> .
	Application Programming Interface	The syntax of the functions and procedures within a module or library.
	ARM Boot Flash Utility	The <i>ARM Boot Flash Utility</i> (BootFU) allows modification of the specific boot flash sector on the system.

ARM Debugger for UNIX	The <i>ARM Debugger for UNIX</i> (ADU) and <i>ARM Debugger for Windows</i> (ADW) are two versions of the same ARM debugger software, running under UNIX or Windows respectively.
ARM Debugger for Windows	See <i>ARM Debugger for Unix</i> .
ARM Developer Suite	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.
ARM eXtensible Debugger	The <i>ARM eXtensible Debugger</i> (AXD) is the latest debugger software from ARM that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target. AXD is supplied in both Windows and UNIX versions.
ARM Flash Utility	The <i>ARM Flash Utility</i> (AFU) is an application for manipulating and storing data within a system that uses the flash library.
ARM Firmware Suite	A collection of utilities to assist in developing applications and operating systems on ARM-based systems.
armsd	The ARM Symbolic Debugger (armsd) is an interactive source-level debugger providing high-level debugging support for languages such as C, and low-level support for assembly language. It is a command-line debugger that runs on all supported platforms.
ARMulator	ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.
ATPCS	The <i>ARM and Thumb Procedure Call Standard</i> (ATPCS) defines how registers and the stack are used for subroutine calls.
AXD	See <i>ARM eXtensible Debugger</i> .
Big-Endian	Memory organization where the least significant byte of a word is at a higher address than the most significant byte. See also <i>Little-endian</i> .
BootFU	See <i>ARM Boot Flash Utility</i> .
Boot monitor	A ROM-based monitor that communicates with a host computer using simple commands over a serial port. Typically this application is used to display the contents of memory and provide system debug and self-test functions.
Boot switcher	The boot switcher selects and runs an image in application flash. You can store one or more code images in flash memory and use the boot switcher to start the image at reset.
Canonical Frame Address	In DWARF 2, this is an address on the stack specifying where the call frame of an interrupted function is located.

CFA	See <i>Canonical Frame Address</i> .
CodeWarrior IDE	The development environment for the ARM Developer Suite.
Coprocessor	An additional processor which is used for certain operations. Usually used for floating-point math calculations, signal processing, or memory management.
Debugger	An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.
Double word	A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
DWARF	<i>Debug With Arbitrary Record Format</i> (DWARF) is a format for debug tables.
EC++	A variant of C++ designed to be used for embedded applications.
ELF	Executable and Linking Format
Environment	The actual hardware and operating system that an application will run on.
Execution view	The address of regions and sections after the image has been loaded into memory and started execution.
Flash memory	Nonvolatile memory that is often used to hold application code.
Halfword	A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
Hardware abstraction layer	Code designed to conceal hardware differences between different processor systems.
Heap	The portion of computer memory that can be used for creating new variables.
Host	A computer which provides data and other services to another computer.
ICE	In Circuit Emulator.
IDE	Integrated Development Environment, for example the CodeWarrior IDE in ADS.
Image	A binary execution file loaded onto a processor and given a thread of execution. An image may have multiple threads. An image is related to the processor on which its default thread runs.
Inline	Functions that are repeated in code each time they are used rather than having a common subroutine. Assembler code placed within a C or C++ program.
Input section	Contains code or initialized data or describes a fragment of memory that must be set to zero before the application starts. <i>See also</i> Output section

Interworking	Producing an application that uses both ARM and Thumb code.
Library	A collection of assembler or compiler output objects grouped together into a single repository.
Linker	Software which produces a single image from one or more source assembler or compiler output objects.
Little-endian	Memory organization where the least significant byte of a word is at a lower address than the most significant byte. See also <i>Big-endian</i> .
Load view	The address of regions and sections when the image has been loaded into memory but has not yet started execution.
Local	An object that is only accessible to the subroutine that created it.
Memory management unit	Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.
MMU	See <i>Memory Management Unit</i> .
Multi-ICE	Multi-processor in-circuit emulator. ARM registered trademark.
Output section	Is a contiguous sequence of input sections that have the same Read Only (RO), Read Write (RW), or Zero Initialized (ZI) attributes. The sections are grouped together in larger fragments called regions. The regions will be grouped together into the final executable image. <i>See also Region</i>
PCI	See <i>Peripheral Component Interconnect</i> .
PCS	Procedure Call Standard. <i>See also ATPCS</i>
Peripheral Component Interconnect	An expansion bus used with PCs and workstations.
PIC	Position Independent Code. <i>See also ROPI</i>
PID	Position Independent Data. <i>See also RWPI</i>
Profiling	Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code.

Call-graph profiling provides great detail but slows execution significantly. *Flat profiling* provides simpler statistics with less impact on execution speed.

For both types of profiling you can specify the time interval between statistics-collecting operations.

Program image	See Image.
Reentrancy	The ability of a subroutine to have more than one instance of the code active. Each instance of the subroutine call has its own copy of any required static data.
Region	In an Image, a region is a contiguous sequence of one to three output sections. Output section types are Read Only (RO), Read Write (RW), and Zero Initialized (ZI).
Remapping	Changing the address of physical memory or devices after the application has started executing. This is typically done to allow RAM to replace ROM once the initialization has been done.
Retargeting	The process of moving code designed for one execution environment to a new execution environment.
ROPI	Read Only Position Independent. Code and read-only data addresses can be changed at run-time.
RTOS	Real Time Operating System.
RWPI	Read Write Position Independent. Read/write data addresses can be changed at run-time.
Scatter loading	Assigning the address and grouping of code and data sections individually rather than using single large blocks.
Scope	The accessibility of a function or variable at a particular point in the application code. Symbols which have global scope are always accessible. Symbols with local or private scope are only accessible to code in the same subroutine or object.
Section	A block of software code or data for an Image. <i>See also Input section</i>
Semihosting	A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather than attempting to support the I/O itself.
SIB	See <i>System Information Block</i> .
System Information Block	A block of user-defined nonvolatile storage.
SWI	Software Interrupt. An instruction that causes the processor to call a programmer-specified subroutine. Used by ARM to handle semihosting.

Target	<p>The actual target processor, real or simulated, on which the application is running.</p> <p>The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors.</p>
Thread	<p>A context of execution on a processor. A thread is always related to a processor and may or may not be associated with an image.</p>
Veneer	<p>A small block of code used with subroutine calls when there is a requirement to change processor state or branch to an address that cannot be reached from the current processor state.</p>
Watchpoint	<p>A location within the image which will be monitored and which will cause execution to break when it changes.</p>
Word	<p>A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.</p>

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.

A

AFS

- about 1-2
- Angel 1-16
- board demonstrations 1-14
- flash memory and 1-7
- generic demonstrations 1-13
- PCI 1-11
- µHAL 1-2

Angel

- flash 3-14
- verifying 6-4

API

- extended HAL functions 2-3
- simple µHAL functions 2-3
- types 2-3
- µHAL functions 2-3
- µHAL naming conventions 2-5

Applications programming interface,
see API

ARM support x

AXD 1-12

B

Boards

- displaying details 4-2
- supported 6-3

Boot image

- selecting 3-19

Boot monitor

- terminal configuration 3-17

Boot switcher

- flash 6-5

Building

- C library 6-2
- LED demo 3-4
- using GNU make 2-7
- with ADS 6-2
- µHAL application 2-7
- µHAL library 2-9

C

CodeWarrior IDE 2-9

- copy project 5-4
- directories 5-10
- LED demo 3-5
- new project 5-3
- settings 5-12

D

DCC 1-12

Demonstrations 1-14

Directory

- CodeWarrior IDE 5-10
- new 5-2

E

ELF support 6-7

Extended µHAL functions 2-3

F

- Feedback x
- Firmware 1-4
- Flash image
 - Multi-ICE 3-10
- Flash utility 3-15
- Frequently asked questions, μ HAL 2-2
- Further reading viii

G

- Global variables, HAL 2-6
- GNU makefiles 2-9

H

- Host controller 1-12

I

- Image
 - loading 3-18
 - setting boot image 3-19
- Initializing
 - system 2-4
- Installing GNU make on Windows 2-8
- Internal pointers, μ HAL 2-6
- Internal variables, μ HAL 2-6
- Interrupt
 - timer demo 4-2
 - μ HAL control 2-5

L

- LED
 - demo application 3-2
 - μ HAL generic 2-4
- Library
 - build options 5-5
- Loading image
 - Angel 3-15
 - Multi-ICE 3-10

M

- Makefile
 - GNUmake 2-7
 - include 5-7
 - soft link 2-7
 - using 5-13
- Memory
 - management by μ HAL 2-5
- Multi-ICE
 - failure 6-6
 - using 3-7

N

- Naming conventions
 - μ HAL 2-5
 - μ HAL global variables 2-6
 - μ HAL internal variables 2-6
 - μ HAL pointers 2-6

O

- Object types, μ HAL 2-6
- Operating modes, μ HAL applications 2-3

P

- Pointer, μ HAL 2-6
- Projects
 - distributing 6-3

R

- RAM image
 - Angel 3-15
- RDI 1-12
- RealMonitor. *See* RM.
- Related publications viii
- Remote Debug Interface. *See* RDI.
- RM
 - host controller 1-12
 - overview 1-12
 - target functionality 1-12

- RM protocol 1-12
- RTOS 1-12

S

- Serial port
 - μ HAL initialization 2-4
- Simple μ HAL functions 2-3
- S-record
 - boot monitor 3-18
- SWI calls 2-4
- SWI interface
 - chaining 1-11
- System initialization, μ HAL 2-3

T

- Target functionality 1-12
- Terminal
 - configuring 3-17
- Thumb support 6-2, 6-6
- Timer
 - demo 4-2
 - initializing 4-3
 - installing 4-3
 - installing interrupt 4-3
 - requesting 4-3
 - value 4-4
 - μ HAL generic 2-4
- Troubleshooting 6-1
- Typographical conventions vii

U

- uHALr_InitInterrupts() 4-3
- uHALr_InitTimers() 4-3
- uHALr_InstallSystemTimer() 4-3
- uHALr_RequestSystemTimer() 4-3

V

- Vectors
 - chaining 1-11

Symbols

μCOS-II 1-3

μHAL

- application operating modes 2-3

- building 2-9

- frequently asked questions 2-2

- global variable naming 2-6

- internal variable naming 2-6

- licensing 2-2

- naming conventions 2-5

- object types 2-6

- overview 2-2

- pointer naming 2-6

- system initialization 2-3

