ARM Firmware Suite

Version 1.1

Reference Guide

ARM DUI 0102C

Release Information

Change History

Date	Issue	Change
8 September 1999	A	New document (internal release)
10 September 1999	B	First release
15 February 2000	C	Second release

Proprietary Notice

Contents
ARM Firmware Suite

Preface

This preface introduces the ARM Firmware Suite and its reference documentation. It contains the following sections:

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, and how to use them 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.

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 µHAL*

Read this chapter for a description of µHAL (pronounced *Micro-HAL*), a *Hardware Abstraction Layer* that is designed to conceal hardware differences between different ARM-based systems.

Chapter 3 *µHAL Application Programming Interfaces*

Read this chapter for information about the µHAL applications programming interface. This chapter describes the µHAL parameter types and functions.

Chapter 4 *ARM Boot Monitor*

Read this chapter for a description of the boot monitor. This chapter describes how the boot monitor functions and describes its command-line interface.

Chapter 5 *Operating Systems and HAL*

Read this chapter for a description how operating systems are ported to a platform which has µHAL ported to it.

Chapter 6 *Angel*

Read this chapter for a description of how the Angel debug monitor and µHAL are related.

Chapter 7 *Flash Library Specification*

Read this chapter for reference information about the flash library. This chapter discusses the approach to flash management used and describes the firmware flash library functions.

Chapter 8 *Using the ARM Flash Utilites*

Read this chapter for information about using the *ARM Flash Utility* (AFU) and *Boot Flash Utility* (BootFU).

Chapter 9 *PCI Management Library*

Read this chapter for information about PCI management. This chapter describes how PCI resources are initialized and managed, and describes the PCI management functions.

Chapter 10 *Troubleshooting and Frequently Asked Questions*

Read this appendix for answers to common questions or problems.

Chapter 11 *Building AFS Components*

Read this appendix for a description of how to rebuild ARM firmware suite components.

Typographical conventions

The following typographical conventions are used in this book:

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

`type`writer Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the whole command or option name.

`typewriter italic`

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

italic Highlights important notes, introduces special terminology, denotes 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.

`typewriter bold`

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 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 122A)

The following publications provide reference information about ARM architecture:

- *AMBA Specification* (ARM IHI 0011)
- *ARM Architectural Reference Manual* (ARM DDI 0100).

The following publications provide information about the ARM Software Development Toolkit 2.5:

- *ARM Software Development Toolkit User Guide* (ARM DUI 0040)
- *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

The following publications provide information about the ARM Developer Suite 1.0:

- *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 CE is available from the Microsoft web site:

`http://www.microsoft.com`

The following publication provides information about µC/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 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:

1.1 What is firmware?

Firmware is low-level software that runs on evaluation 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 provides a collection of standard functions with a known *Application Programming Interface* (API). This API enables applications to perform hardware-specific operations without requiring a version of the application for each hardware configuration.

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.

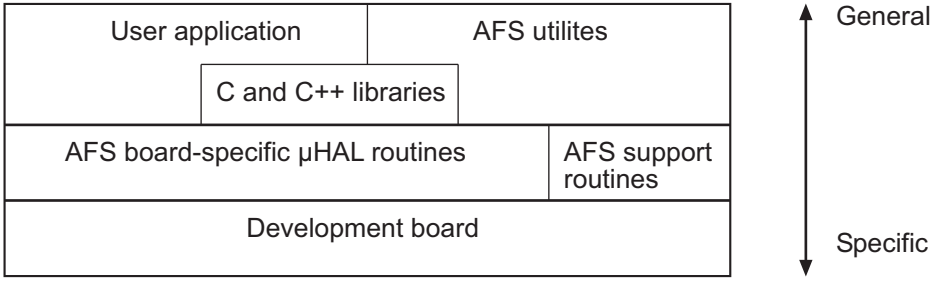

Figure 1-1 Logical organization

1.2 About the ARM Firmware Suite

AFS provides:

µHAL libraries

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

Flash libraries

The flash library provides an 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 have an easy to use interface that simplifies 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 ADS 1.0 and SDT 2.5, and it supports 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. For example, the supplied PCI library supports the PCI bus on the Integrator board.

Angel A version of Angel that has been implemented using µHAL is included with AFS.

µC/OS-II AFS includes a port of µC/OS made to the ARM architecture using the µHAL interfaces.

1.2.1 µHAL Libraries

µHAL masks hardware differences between platforms by providing a standard layer of board-dependent functions for I/O, RAM, boot flash, and application flash. The µHAL 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 or PCMCIA
- identify and initialize system devices, for example serial interfaces
- initialize and handle interrupts
- access code or data stored in flash memory.

The µHAL API consists of two types of function:

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 µHAL fits together and how it works. Within µHAL itself, the basic functions are built using the extended functions.

Examples of specific functional modules in µHAL are:

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

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

You can write µHAL 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, accesses to these facilities are requested by using *Software Interrupt Instructions* (SWIs).

The implementation of µHAL is described in Chapter 2 *An Introduction to µHAL*, and using the µHAL applications programming interface is described in Chapter 3 *µHAL Application Programming Interfaces*.

1.2.2 Flash libraries 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 AFS utilities to load applications into the flash or RAM memory.

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 Chapter 7 *Flash Library Specification*.

The ARM Flash Utility

The *ARM Flash Utility* (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 Debugger for Windows* (ADW) environment. 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.

Using the utilities is described in Chapter 8 *Using the ARM Flash Utilites*.

The ARM Boot Flash Utility

With the *ARM Boot Flash Utility* (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 *ARM Debugger for Windows* (ADW) environment.

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.

See Chapter 4 *ARM Boot Monitor* for more information on the boot monitor and boot switcher.

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 the your own programs into the application flash.

Boot Boot flash holds the boot monitor and boot switcher utilities used for loading and debugging applications.

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 on page 1-8. 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 non-volatile storage area for various processes.

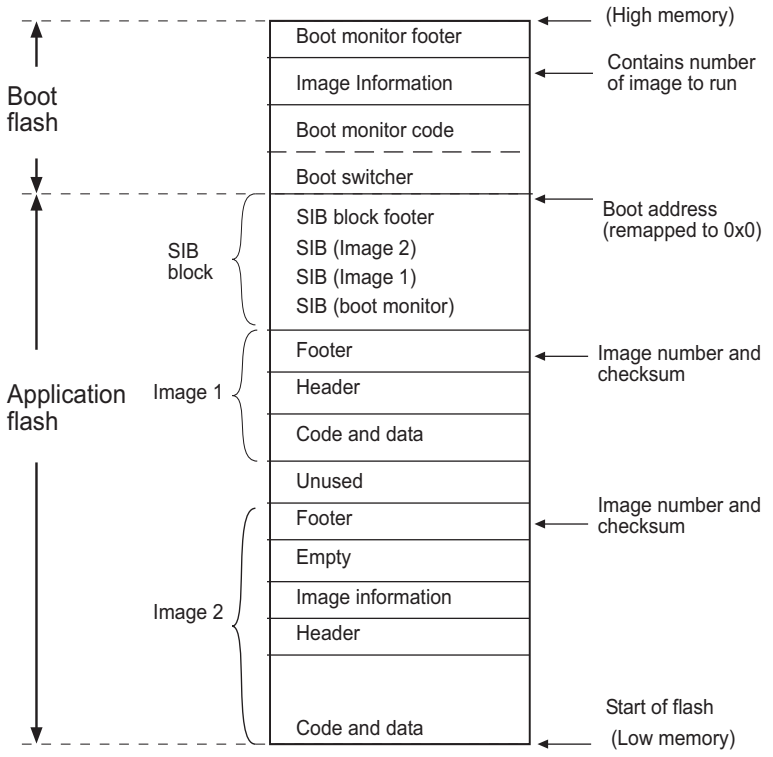

Figure 1-2 Images in flash

1.2.3 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 Chapter 4 *ARM Boot Monitor* for more information on the boot monitor.

1.2.4 Generic µHAL demonstration programs

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

- *Simple tests*
- *Timing tests* on page 1-10.

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:

`hello.c` This program outputs data to the serial port.

`io.c` This program takes data input on the serial port and echoes it on the output.

`led.c` 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, since this functionality is known to be available in semihosted mode.

`heap.c` This program allocates and then frees some memory.

`simple-caches.c`

This program gives an example of simple cache (Data and Instruction) usage that:

- reads the cache and MMU state
- resets the cache and MMU
- turns caches and MMU on and off
- restores the original state.

`system-timers.c`

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.

`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 only runs 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.

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 regimes. 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 placed on a 20 by 20 chess board so that no queens checks 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.

`sieve500.c`

This is a repeat of the `sieve.c` test, however the main loop is unrolled to guarantee that the instruction cache is ineffective.

Note

There are no build files for `sieve500`. Use the build files for `sieve` as a starting point.

1.2.5 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. 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.

`irda` Infrared transmit and receive applications.

`keyboard` Application to scan the keyboard and display key codes.

1.2.6 PCI management libraries

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

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 Chapter 9 *PCI Management Library*.

1.2.7 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 the ARM state 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.

Using Angel is described in Chapter 6 *Angel*.

1.2.8 µC/OS-II

µC/OS-II is a portable, ROM-able, preemptive, real-time, multitasking kernel that can manage up to 63 tasks. µC/OS-II is comparable in performance to many commercially available kernels. AFS includes a port of µC/OS made to the ARM architecture using the µHAL interfaces.Operating systems are described in Chapter 5 *Operating Systems and HAL*.

Chapter 2
An Introduction to µHAL

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

2.1 About µHAL

ARM processor cores are highly integrated and are used in a wide range of products. The ARM processor core is often the only common feature shared by these products. µHAL consists of a low-level interface that provides a common set of functions for these different ARM-based systems.

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

µHAL 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.

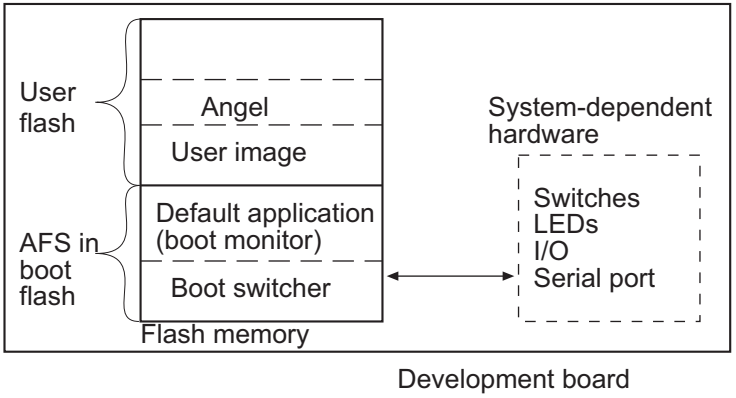

Figure 2-1 Development board with AFS

2.1.1 Licensing

µHAL 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 µHAL in commercial projects, contact ARM to ensure that you have the appropriate µHAL 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

The answers to some common setup problems and frequently asked questions can be found in Chapter 10 *Troubleshooting and Frequently Asked Questions*.

The ARM 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 µHAL 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 µHAL components fit together and how it works. Within µHAL itself, the basic functions are built using the extended functions.

For a complete list and description of these functions, see Chapter 3 *µHAL Application Programming Interfaces*.

2.1.4 Application operating modes

µHAL applications can be written 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, accesses to these facilities are requested by using *Software Interrupt 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 µHAL

µHAL 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 2-5
- *Interrupt control* on page 2-5
- *Memory management* on page 2-5

System initialization software

This software initializes the system so that the standard µHAL 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.

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, µHAL 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, µHAL makes SWI calls to the debug host to run `printf()`.

Generic timer

µHAL provides a generic interface to manage any timers present in the system. In semihosted mode, µHAL 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, µHAL provides a generic interface to these LEDs.

Interrupt control

µHAL v1.1 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.

µHAL v1.1 assumes that it has complete control of the IRQs. If the application is semihosted, the debug agent can use the *Fast Interrupt Requests* (FIQs).

Memory management

If the system has not been set up by a debug agent, µHAL 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 systems based on SA-1100, 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.

2.1.6 µHAL naming conventions

Every µHAL v1.1 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`

Table 2-1 µHAL naming conventions (continued)

Object type	Sample
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

You must have an understanding of how µHAL operates internally before you use the extended routines.

2.2 Building a new µHAL-based application

There are three ways to build a µHAL-based application:

- Take an existing demonstration program (for example `uHALDemos/Sources/hello.c`) and modify it for your use. You can use the program with any of the existing build systems.
- Create a new component and build your software there with one of the build systems. You must understand the details of the build system that you are using.
- Copy the prebuilt µHAL 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 µHAL 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.

2.3 Building the µHAL library

There are three ways of building the µHAL library and other AFS components:

- using ARM `.mcp` project files for the CodeWarrior IDE with ADS 1.0
- using ARM `.apj` project files for the ARM Project Manager IDE with SDT 2.5
- using GNU makefiles.

Each board and processor combination has its own build directory within `uHAL\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:

```
windows\source\Integrator940T\uHAL\Build\Integrator940T.b.
```

The project files and makefiles in `Integrator940T.b` build two variants of the µHAL 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 `uHALLibrary_Data` and creates the `standalone` and `semihosted` subdirectories in the new directory.

See Chapter 11 *Building AFS Components* for details on rebuilding components.

Chapter 3
µHAL Application Programming Interfaces

This chapter describes the simple and extended APIs to µHAL. It contains the following sections:

3.1 About the µHAL APIs

This section provides an overview of the general APIs provided by µHAL. See *µHAL PCI function descriptions* on page 9-16 for a description of PCI functions contained in µHAL.

3.1.1 µHAL-specific function types

µHAL uses three function types that are abstracted to make interface routines easier to use. These are described in Table 3-1.

Table 3-1 Parameter types

Description	Syntax
A pointer to a function with no argument. The function does not return a value.	`typedef void (*PrVoid)(void);`
A pointer to a function with one integer argument. The function does not return a value.	`typedef void (*PrHandler)(unsigned int);`
A pointer to a function with no argument. The function returns a `PrVoid` pointer to a function.	`typedef PrVoid (*PrPrVoid)(void);`

For example, with the `uHALr_RequestTimer()` declaration:

```
int uHALr_RequestTimer(PrHandler handler,
                       const unsigned char *devname)
```

an interrupt handler can be declared as:

```
void TickTimer(unsigned int interrupt)
```

and registered with µHAL using:

```
uHALr_RequestSystemTimer(TickTimer, "test");
```


3.1.2 Simple and extended API functions

Using the µHAL simple API does not require an understanding of how µHAL works, or of the ARM architecture.

Using the µHAL extended API requires an understanding of how µHAL functions. All functions and type definitions are contained in `h\cdefs.h` and `h\uhal.h`.

Note

A number of demonstration programs that use this interface can be found in the `uHALDemos` subdirectory of the AFS installation. The code examples used in this section are taken from these demonstration programs.

3.2 Simple API memory functions

This section describes the set of functions that are used to find free memory in the system, and to allocate and free heap storage. Free memory is memory that is not used by µHAL itself or a debug agent. Prototypes for all of these functions are available in `h\uhal.h`.

The memory functions are:

- *uHALr_StartOfRam()*
- *uHALr_EndOfFreeRam()*
- *uHALr_EndOfRam()* on page 3-5
- *uHALr_HeapAvailable()* on page 3-5
- *uHALr_InitHeap()* on page 3-5
- *uHALr_malloc()* on page 3-6
- *uHALr_free()* on page 3-6.

There is an example of a program that allocates and de-allocates heap storage in *Example of heap allocation and de-allocation* on page 3-7.

3.2.1 uHALr_StartOfRam()

This function returns the address of the first free uninitialized RAM location.

Syntax

```
void *uHALr_StartOfRam(void)
```

Return value

Returns the `void*` address of the first available RAM location.

3.2.2 uHALr_EndOfFreeRam()

This function returns the address of the last available RAM location.

Syntax

```
void *uHALr_EndOfFreeRam(void)
```

Return value

Returns the `void*` address of the last available RAM location.

3.2.3 uHALr_EndOfRam()

This function returns the address of the last RAM location.

Syntax

```
void *uHALr_EndOfRam(void)
```

Return value

Returns a pointer to the address of last RAM location.

3.2.4 uHALr_HeapAvailable()

This function returns a flag to indicate whether this port of the µHAL library includes support for heap management.

Syntax

```
int uHALr_HeapAvailable(void)
```

Return value

Returns one of the following:

1 If the heap management functions are included in the library.

0 If heap management functions are not included.

3.2.5 uHALr_InitHeap()

This function initializes the heap. It must be called before any memory allocation or de-allocation is attempted.

Syntax

```
void uHALr_InitHeap(void)
```

3.2.6 uHALr_malloc()

This function allocates contiguous storage from the heap.

Syntax

```
void *uHALr_malloc(unsigned int size)
```

where:

size is the number of bytes of memory required.

Return value

Returns

0	If size was 0.
-1	If the memory could not be allocated.
pointer	If successful, the `void*` address of allocated memory.

3.2.7 uHALr_free()

This routine frees previously allocated memory pointed at by *memPtr*.

Syntax

```
void uHALr_free(void *memPtr)
```

where:

memPtr is a pointer to the heap memory to be freed. This value should not be –1. If the value is 0, the function returns without taking any action.

3.2.8 Example of heap allocation and de-allocation

Example 3-1 shows an example of a program that allocates and de-allocates heap storage. The program can be found in `uHALDemos\Sources\heap.c`.

Example 3-1 Allocating and de-allocating heap storage

```
#include "uhal.h"
int main (int argc, int *argv[])
    {
        int Ch ;
        int i ;
        void *memP ;
        uHALr_printf("*** HEAP Allocation/Deallocation ***\n") ;
        // init
        uHALr_InitHeap() ;
        // allocate and free some memory
        for (i = 0 ; i < 16 ; i++) {
            uHALr_printf("malloc'ing 0x%X bytes...", i * 16) ;
            memP = uHALr_malloc(i * 16) ;
            uHALr_printf("@ 0x%X\n", memP) ;
            uHALr_free(memP) ;
    }
    return (OK);
}
```

3.3 Simple API interrupt functions

µHAL assumes that interrupts occur using IRQs. These routines allow you to:

- install a generic interrupt handler
- request control of a particular interrupt
- enable and disable that interrupt.

Your application can install different interrupt handlers for different interrupts, or install a single handler for many interrupts.

When an interrupt occurs, µHAL traps it and calls the appropriate handler routine, passing it the number of the interrupt that occurred.

Note

µHAL does not provide any support to the application for finding the source of interrupts. It is the responsibility of the board-specific code to map the programmable interrupt controller format to and from a 32-bit quantity.

The interrupt functions are:

- *uHALr_InitInterrupts()*
- *uHALr_RequestInterrupt()* on page 3-9
- *uHALr_FreeInterrupt()* on page 3-9
- *uHALr_EnableInterrupt()* on page 3-10
- *uHALr_DisableInterrupt()* on page 3-10.

3.3.1 uHALr_InitInterrupts()

This function is called once on startup by the application. It initializes the µHAL internal interrupt structures. This must be called before installing a new IRQ handler.

Syntax

```
void uHALr_InitInterrupts(void)
```

3.3.2 uHALr_RequestInterrupt()

This function assigns a high-level handler routine to the specified interrupt. It sets up the internal structures, but does not activate the interrupt.

Syntax

```
int uHALr_RequestInterrupt(unsigned int intNum,
                PrHandler handler,
                const  unsigned char *devname)
```

where:

intNum is the number of the interrupt to be processed.

handler is a pointer to the routine that processes the interrupt.

devname is a pointer to a string identifying the function of the interrupt.

Return value

Returns one of the following:

0 If successful.

-1 If *intNum* is unknown or already assigned.

3.3.3 uHALr_FreeInterrupt()

This function removes the high-level handler from the specified interrupt.

Note

An application should always call `uHALr_DisableInterrupt()` before calling this routine. Call `uHALr_FreeInterrupt()` before attempting to change the routine associated with an interrupt.

Syntax

```
int uHALr_FreeInterrupt(unsigned int intNum)
```

where:

intNum is the number of the interrupt to be freed.

Return value

Returns one of the following:

0 If successful.

-1 If *intNum* is unknown, reserved, or not allocated.

3.3.4 uHALr_EnableInterrupt()

This function enables the specified interrupt. On many ARM-based systems, this is a two-step process. It enables an on-board interrupt controller, and then it enables the interrupt mask on the processor.

Syntax

```
void uHALr_EnableInterrupt(unsigned int intNum)
```

where:

intNum is the number of the interrupt to be enabled.

3.3.5 uHALr_DisableInterrupt()

This function disables the specified interrupt. On many ARM-based systems, interrupts are enabled and disabled at two stages:

- an on-board controller
- the interrupt mask on the processor.

The `uHALr_DisableInterrupt()` function disables the interrupt on the interrupt controller and does not affect masking by the processor.

Syntax

```
void uHALr_DisableInterrupt(unsigned int intNum)
```

where:

intNum is the number of the interrupt to be disabled. The routine has no effect if the number is not in the range of valid interrupts.

3.4 Simple API MMU and cache functions

On processors that support it, µHAL allows an application to:

- Turn virtual memory on and off using the *Memory Management Unit* (MMU). (On systems with read-only memory at address 0, the MMU cannot be disabled.)
- Enable and disable the caches.

These functions are:

- *uHALr_ResetMMU()*
- *uHALr_InitMMU()*
- *uHALr_EnableCache()* on page 3-12
- *uHALr_DisableCache()* on page 3-12.

Memory management and cache code example on page 3-12 includes an example of a basic cache manipulation program.

3.4.1 uHALr_ResetMMU()

This function safely resets the MMU (and caches) to a fully disabled state (all OFF), irrespective of the state they were originally in. If the MMU cannot be disabled, this function has no effect.

Syntax

```
void uHALr_ResetMMU(void)
```

3.4.2 uHALr_InitMMU()

This function initializes the MMU to a default one-to-one mapping. This mapping also defines the types of access allowed to each area according to execution mode. For example, flash can be written in Supervisor mode, but not User mode.

Syntax

```
void uHALr_InitMMU(int mode)
```

where:

mode is any combination of the MMU mode flags and cache bit flags, `EnableMMU`, `IC_ON`, `DC_ON`, and `WB_ON`. See also *uHALir_WriteCacheMode()* on page 3-43.

3.4.3 uHALr_EnableCache()

This function provides a way to enable all caches that are supported by the processor.

Syntax

```
void uHALr_EnableCache(void)
```

3.4.4 uHALr_DisableCache()

This function disables all caches that are supported by the processor.

3.4.5 Memory management and cache code example

Example 3-2 is an example of a simple cache manipulation program, taken from `uHALDemos\Sources\simple-caches.c`:

Example 3-2 MMU and cache

```
#include "uhal.h"
#include "mmu_h.h"
int main (int argc, int *argv[]) {
    // who are we?
    uHALr_printf("Simple Cache Usage [v1.0]\n") ;
    // First reset the caches to a known state
    uHALr_printf("Resetting caches...") ;
    uHALr_ResetMMU() ;
    uHALr_printf("done\n") ;
    // Now init the MMU to all on
    uHALr_printf("Enabling the MMU and all caches...") ;
    uHALr_InitMMU(IC_ON | DC_ON | WB_ON | EnableMMU) ;
    uHALr_printf("done\n") ;
    // Disable the caches
    uHALr_printf("Disabling all caches...") ;
    uHALr_DisableCache() ;
    uHALr_printf("done\n") ;
    // Finally, enable all of the caches
    uHALr_printf("Enabling all caches...") ;
    uHALr_EnableCache() ;
    uHALr_printf("done\n") ;
    // go home
    return (OK);
}
```


3.5 Simple API timer functions

µHAL provides a set of routines that allow an application to use a timer as a system or operating system timer. This is the simplest way to use timers in µHAL.

µHAL also provides generic timer access routines that give more direct access (although with a little more complexity) to the timers in the system.

The timer functions are:

- *uHALr_CountTimers()*
- *uHALr_InitTimers()* on page 3-14
- *uHALr_RequestSystemTimer()* on page 3-14
- *uHALr_InstallSystemTimer()* on page 3-15
- *uHALr_RequestTimer()* on page 3-16
- *uHALr_InstallTimer()* on page 3-16
- *uHALir_GetSystemTimer()* on page 3-50
- *uHALr_FreeTimer()* on page 3-16
- *uHALr_GetTimerInterval()* on page 3-17
- *uHALr_SetTimerInterval()* on page 3-17
- *uHALr_GetTimerState()* on page 3-18
- *uHALr_SetTimerState()* on page 3-18
- *uHALr_EnableTimer()* on page 3-19.

System timer programming example on page 3-15 shows how to use a system timer.

3.5.1 uHALr_CountTimers()

This function returns the number of timers that are supported by the target.

Syntax

```
unsigned int uHALr_CountTimers(void)
```

Return value

Returns the number of timers supported by the target.

3.5.2 uHALr_InitTimers()

This function must be called before any other timer function. This function

- initializes the µHAL internal interrupt structures
- resets all timers to a known state. (It sets the internal delays to a predefined value and sets all timers off.)

If this function is compiled for use with a debug agent, such as Angel, the timer associated with the debug agent is not reset and is locked to prevent access from within µHAL.

Note

For the timer interrupt handler to be correctly installed, the application must ensure that `uHALr_InitInterrupts()` has been called before this function call.

Syntax

```
void uHALr_InitTimers(void)
```

3.5.3 uHALr_RequestSystemTimer()

This function installs a handler for the system timer, sets up the internal structures, and stops (and does not restart) the timer. By default, the system timer is set to tick once every millisecond.

Syntax

```
int uHALr_RequestSystemTimer(PrHandler handler,
                    const unsigned char *devname)
```

where:

handler is a pointer to the routine that will process the interrupt.

devname is a pointer to a string identifying the function of the interrupt.

Return value

Returns one of the following:

0 If successful.

-1 If the IRQ is already assigned.

3.5.4 uHALr_InstallSystemTimer()

This function starts the timer and enables the interrupt associated with it.

Syntax

```
void uHALr_InstallSystemTimer(void)
```

3.5.5 System timer programming example

The program in Example 3-3 demonstrates how a system timer is used.

Example 3-3 System timer example

```
#include "uhal.h"
// High-level routine called by IRQ Trap Handler when the timer interrupts
static int  OSTick = 0 ;
void TickTimer(unsigned int irq){
    OSTick++ ;
}
    int main (int argc, int *argv[]) {
    int i, j ;

    uHALr_printf("System Timer\n") ; // who are we?
    uHALr_InitInterrupts() ;   // Install new trap handlers and soft vectors
    uHALr_InitTimers() ;            // initialize the timers
    OSTick = 0 ;                    // initialize the tick count
    uHALr_printf("Timer init\n") ;
    if (uHALr_RequestSystemTimer(TickTimer,(const unsigned char*)"test")<= 0)
       uHALr_printf("Timer/IRQ busy\n") ;

    uHALr_InstallSystemTimer() ;     // Start system timer & enable the interrupt
    // 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 is %x\n", OSTick) ;
        for (i = 0 ; i < 1000000 ; i++) ;
    }
    return (OK);
}
```

3.5.6 uHALr_RequestTimer()

This function gets the next available timer and installs a handler. On return, the timer is initialized but not running.

Syntax

```
int uHALr_RequestTimer(PrHandler handler,
                       const unsigned char *devname)
```

where:

handler is a pointer to the routine that will process the interrupt.

devname is a pointer to a string identifying the function of the interrupt.

Return value

Returns one of the following:

timer If successful, the timer number is returned.

-1 If the timer is unknown or already assigned.

3.5.7 uHALr_InstallTimer()

This function starts the specified timer by enabling the timer and the associated interrupt.

```
void uHALr_InstallTimer(unsigned int timer)
```

where:

timer is the timer to be started.

3.5.8 uHALr_FreeTimer()

This function disables the specified timer, frees the interrupt, and updates the internal structure.

Syntax

```
int uHALr_FreeTimer(unsigned int timer)
```

where:

timer is the number of the timer to be freed.

Return value

Returns one of the following:

0	If successful.
-1	If the timer is unknown.

3.5.9 uHALr_GetTimerInterval()

This function gets the interval, in microseconds, for the specified timer.

Syntax

```
int uHALr_GetTimerInterval(unsigned int timer)
```

where:

timer	is the number of the timer for which the interval is requested.

Return value

Returns one of the following:

interval	If successful (return value in microseconds).
-1	If the timer is not found.

3.5.10 uHALr_SetTimerInterval()

This function sets the interval, in microseconds, for the specified timer.

Syntax

```
int uHALr_SetTimerInterval(unsigned int timer,
                           unsigned int interval)
```

where:

timer	is the timer number for which the interval is to be set.
interval	is the number of microseconds between events.

Return value

Returns one of the following:

0	If the timer is found.
-1	If the timer is not found.

3.5.11 uHALr_GetTimerState()

This function gets the current state of the specified timer.

Syntax

```
int uHALr_GetTimerState(unsigned int timer)
```

where:

timer is the timer number for which the state is requested.

Return value

Returns one of the following:

state If the timer is found, the current state is one of:

T_FREE	Available.
T_ONESHOT	Single-shot timer (in use).
T_INTERVAL	Repeating timer (in use).
T_LOCKED	Not available for µHAL.

-1 If the timer is not found.

3.5.12 uHALr_SetTimerState()

This function sets the timer state.

Syntax

```
int uHALr_SetTimerState(unsigned int timer,
                        enum uHALe_TimerState state)
```

where:

timer is the timer number for which the state is being set.

state is a valid timer state which is one of:

T_ONESHOT	Single-shot timer (in use).
T_INTERVAL	Repeating timer (in use).

Return value

Returns one of the following:

0 If the timer is found.

-1 If the timer is not found.

3.5.13 uHALr_EnableTimer()

This function reloads the interval and enables the specified timer.

Syntax

```
void uHALr_EnableTimer(unsigned int timer)
```

where:

timer is the timer to be enabled.

3.6 Simple API support functions

In addition to the general routines, µHAL provides implementations of a number of standard C library routines. The support functions include:

- *uHALr_memset()*
- *uHALr_memcmp()*
- *uHALr_memcpy()* on page 3-21
- *uHALr_strlen()* on page 3-21.

3.6.1 uHALr_memset()

This function places character *c* into the first *n* characters of *s*, and returns *s*.

Syntax

```
void *uHALr_memset(char *s, int c, int n)
```

where:

s	is the start address of memory to be set.
c	is the character to be copied into memory.
n	is the number of memory locations to be used.

Return value

Returns *s*.

3.6.2 uHALr_memcmp()

This function compares the first *n* characters of *cs* with *ct*.

Syntax

```
int uHALr_memcmp(char *cs, char *ct, int n)
```

where:

cs	is the start of memory locations to be compared.
ct	is the start of memory locations to be compared *against*.
n	is the number of memory locations to be compared.

Return value

Returns one of the following:

1	If *cs*>*ct*.
0	If *cs*=*ct*.
-1	If *cs*<*ct*.

3.6.3 uHALr_memcpy()

This function copies *n* characters from *ct* to *s*.

Syntax

```
int uHALr_memcpy(char *s, char *ct, int n)
```

where:

s	is a pointer to the destination memory locations.
ct	is a pointer to the source memory locations.
n	is the number of memory locations to be copied.

Return value

Returns the address of the first location copied.

3.6.4 uHALr_strlen()

This function returns the length of *s*.

Syntax

```
int uHALr_strlen(const char *s)
```

where:

s	is a pointer to a zero-terminated string.

Return value

This function returns the size, in bytes, of *s*.

3.7 Simple API LED control functions

µHAL provides a set of simple routines for accessing any LEDs in the system. The LED control functions are:

- *uHALr_CountLEDs()* on page 3-23
- *uHALr_InitLEDs()* on page 3-23
- *uHALr_ResetLED()* on page 3-23
- *uHALr_SetLED()* on page 3-24
- *uHALr_ReadLED()* on page 3-24
- *uHALr_WriteLED()* on page 3-24.

An example of a simple LED flashing program is provided in *LED control code example* on page 3-25.

3.7.1 LED states and addresses

The µHAL LED code is generic and manages any LEDs that can be accessed at different addresses on different boards. Logic 1 can indicate either ON or OFF.

The LED code in the module `Sources\led.c` keeps the LED addresses (or homes) in the `uHALiv_LedHomes` array. The set of pointers to LEDs is initialized to be the contents of `uHAL_LED_OFFSETS`. The addresses, pointers, and the number of LEDs (`uHAL_NUM_OF_LEDS`), are defined in the board-specific definition files `platform.s` and `platform.h`.

For some systems, the platform files contain different addresses for different LEDs. The LED code also keeps a set of masks, one per LED, in the `uHALiv_LedMasks` array. This is set to the contents of `UHAL_LED_MASKS`.

When reading the LEDs, the LED code does the following:

1. Reads the LED using its home address.
2. ANDs the value read with the mask for this LED.
3. Compares the result with the board-specific literal `uHAL_LED_ON`. Some LEDs report 0 as on.A board-specific LED write function, `uHALr_WriteLED()` in `board.c`, is used to write to the LEDs.

3.7.2 uHALr_CountLEDs()

This function returns the number of LEDs available to the µHAL application.

Syntax

```
unsigned int uHALr_CountLEDs(void)
```

Return value

Returns the number of LEDs:

0	If there are no LEDs.
count	If there are LEDs.

3.7.3 uHALr_InitLEDs()

This function initializes the LEDs in the system to `OFF`.

Syntax

```
unsigned int uHALr_InitLEDs(void)
```

Return value

Returns the number of LEDs

3.7.4 uHALr_ResetLED()

This function turns the specified LED off.

Syntax

```
void uHALr_ResetLED(unsigned int led)
```

where:

led	is the specified LED number.

3.7.5 uHALr_SetLED()

This function turns the specified LED on.

Syntax

```
void uHALr_SetLED(unsigned int led)
```

where:

led is the specified LED number.

3.7.6 uHALr_ReadLED()

This function returns the state of the specified LED.

Syntax

```
int uHALr_ReadLED(unsigned int led)
```

where:

led is the specified LED number.

Return value

Returns one of the following:

TRUE If the LED state is on.

FALSE If the LED state is off.

-1 If the LED number specified is invalid.

TRUE is defined as 1 and FALSE is defined as 0.

3.7.7 uHALr_WriteLED()

This function writes a value to the specified LED.

Syntax

```
int uHALr_WriteLED(unsigned int led, unsigned int state)
```


where:

led is the specified LED number.

state is the desired LED state:

TRUE to turn the led on (1).

FALSE to turn the led off (0).

Return value

Returns one of the following:

0 If successful.

-1 If the LED number specified is invalid.

3.7.8 LED control code example

Example 3-4 is an example of a simple LED flashing program (taken from `uHALDemos\Sources\led.c`).

Example 3-4 LED flashing program

```
#include "uhal.h"
int main (int argc, int *argv[])
{
    unsigned int count, max, on ;
    unsigned int wait, i, j ;
    count = uHALr_InitLEDs() ;
    max = (1 << count) ;
    while(1) {
        for (i = 0 ; i < max ; i++ ) {
        /* which LEDs are on? */
        on = (max - 1) & i ;
        for (j = 0; j < count ; j++)
            if (on & (i << j )
                uHALr_SetLED( j + 1 );
            else
                uHALResetLED (j + 1);
            /* wait a while */
            for (wait = 0 ; wait < 1000000 ; wait++) ;
        }
    }
    return (OK);
}
```

3.8 Serial input/output functions, definitions, and macros

If there is a serial port, µHAL provides access for the application by using a series of polled calls. In the case of a semihosted application, µHAL makes SWI calls to the underlying debug agent to process the requests.

The simple serial I/O functions are:

- *uHALr_ResetPort()*
- *uHALr_getchar()*
- *uHALr_putchar()* on page 3-27
- *uHALr_printf()* on page 3-27.

A basic character I/O program example is provided in:

- *Serial input/output code example* on page 3-28.

The one extended serial function is:

- *uHALir_InitSerial()* on page 3-27.

3.8.1 uHALr_ResetPort()

This function resets the port defined for `stdin`/`stdout` to the board default state.

Syntax

```
void uHALr_ResetPort(void)
```

3.8.2 uHALr_getchar()

This function waits for a character from the default port. When compiled as a semihosted application, this function uses the SWI handler provided by the debug agent to get the character from the host console.

Syntax

```
unsigned int uHALr_getchar(void)
```

Return value

Returns the **`unsigned int`** containing the character read from the serial port.

3.8.3 uHALr_putchar()

This function sends the given character to the default port. When compiled as a semihosted application, this function uses the SWI handler provided by the debug agent to send the character to the host console.

Syntax

```
void uHALr_putchar(unsigned char c)
```

where:

c is the character to be sent to the serial port.

3.8.4 uHALr_printf()

This function converts, formats, and writes the arguments to the standard output.

Syntax

```
void uHALr_printf(char *format, ...)
```

where:

format is a pointer to the start of the zero-terminated formatting string. The known format types are:

`%i, %c, %s, %d, %u, %o, %x`, and `%X`

You must insert one of these parameters into the format string.

`...` is a variable list of arguments to print.

3.8.5 uHALir_InitSerial()

This function initializes the specified port to the specified baud rate.

Note

This is an extended API function.

Syntax

```
void uHALr_InitSerial(unsigned int port, unsigned int baudRate)
```

where:

port	is the base address of the serial port to be initialized.
baudRate	is the platform-specific value used to set the data transfer rate.

3.8.6 Serial input/output code example

Example 3-5 is an example of a program performing simple character I/O.

Example 3-5 Serial interface program

```
#include "uhal.h"
extern void print_header( void);
char * test_name = "Input/Output Tests\n";
char * test_ver = "Program Version 1.0\n";
extern void print_end (void);

static int yesno(char *question, int preferred) {
  int c ;
  uHALr_printf(question) ; // ask the question
  if (preferred)
    uHALr_printf("[Yn]? ") ;
  else
    uHALr_printf("[Ny]? ") ;
  c = uHALr_getchar() ; // get the answer and interpret it
  uHALr_putchar(c) ;
  if (c == '\n') return preferred ;
  uHALr_putchar('\n') ;
  return ((c == 'y')||(c == 'Y')) ;
}
int main (int argc, int *argv[]) {
  int i ;
  char buf[80] ;
  U8 c ;
  print_header();    // who are we?
  // Ask for some characters (don't forget to echo)
  uHALr_printf("Please enter a string terminated by C/R\n") ;
  uHALr_printf("IO> ") ;
  for (i = 0 ; i < sizeof(buf) ; i++) {
    c = uHALr_getchar() ;
    uHALr_putchar(c) ;
    if ((c == '\n') || (c == '\r')) {
      uHALr_putchar('\n') ;
      break ;
    }
  }
```

```
 // ask the user if they saw it correctly
  if (yesno("Were the characters echoed to screen properly", 1) == 1)
    uHALr_printf("Successful!\n") ;
  else
    uHALr_printf("Failed!\n") ;

  print_end ();
  return (OK);
}
```

3.8.7 Serial input output board-specific definitions and macros

If µHAL is built to run as a semihosted application, all input and output is handled by the debug agent, for example Angel. In standalone mode, µHAL provides minimal serial input and output support, enough to reset the defined serial port and to handle polled input and output.

The board-specific definition files, `platform.s` and `platform.h`, describe the COM ports for a system and their usage. Example 3-6 on page 3-29 shows the COM port definitions for an SA-1100 Prospector board.

Example 3-6 COM port definitions

```
/* define it so that it only ever uses one port */
/* Default port to talk to host (via debugger) */
#define HOST_COMPORT  UART3_BASE
#define OS_COMPORT    HOST_COMPORT
```

where:

`HOST_COMPORT`
: is the COM port used to communicate with a debug host using the Angel debug monitor.

`OS_COMPORT`
: is the COM port used by an operating system or µHAL application.

On the Prospector board, these are defined to be the same so that a semihosted µHAL application uses semihosting for serial input and output. Because the Prospector board has two COM ports, you can use separate ports to prove that your application works using a real serial port. The board must supply a COM port-specific reset function, `uHALir_InitSerial()`. This can be found in the board-specific `board.c` module.

Note

If you are using Multi-ICE with a semihosted application, the COM port is still reserved. Change the definition in `platform.h` to free the port.

The µHAL COM port input and output code is found in the module `Sources\lib\iolib.c`. It makes use of several macros (defined in the board-specific definition files) to perform polled character input and output.

These macros are:

- `GET_CHAR`
- `GET_STATUS`
- `RX_DATA`
- `TX_READY`
- `PUT_CHAR`.

These macros support the µHAL input/output functions such as `uHALr_PutChar()`. Example 3-7 shows macros for the Prospector.

Example 3-7 Prospector usage of serial input/output macros

```
* This board uses the SA-1100 UART3 as stdio */
#define UART3_BASE                      0x80050000

/* UART primitives */
#define PUT_CHAR(p, c)   ((*(volatile unsigned int *)
                            (p + UTDR)) = c)
#define GET_STATUS(p)    (*(volatile unsigned int *)(p + UTSR1))
#define GET_CHAR(p)      (*(volatile unsigned int *)(p + UTDR))
#define RX_DATA(s)       (s & UTSR1_RNE)
#define TX_READY(s)      ((s & UTSR1_TNF) != 0)
#define RX_ENABLE        0x09
#define TX_ENABLE        0x12
#define TX_BUSY(s)       (s & UTSR1_TBY)
#define READ_INTERRUPT   (p)(*(volatile unsigned int *)
                            (p + UTSR0))
#define RX_INTERRUPT     2
#define TX_INTERRUPT     1
```

```
/* UART regs/values */
#define UTCR0       0x00
#define UTCR1       0x04
#define UTCR2       0x08
#define UTCR3       0x0C
#define UTDR        0x14
#define UTSR0       0x1C
#define UTSR1       0x20

/* Line status bits. */
#define UTSR1_TBY   1   /*  transmitter busy flag */
#define UTSR1_RNE   2   /*  receiver not empty (LSR_DR) */
#define UTSR1_TNF   4   /*  transmit fifo non full */
#define UTSR1_PRE   8   /*  parity read error (LSR_PE) */
#define UTSR1_FRE   16  /*  framing error (LSR_FE) */
#define UTSR1_ROR   32  /*  receive fifo overrun (LSR_OE) */
#define UTSR0_TFS   1   /*  transmit fifo service request */
#define UTSR0_RFS   2   /*  receive fifo service request */
#define UTSR0_RID   4   /*  receiver idle */
#define UTSR0_RBB   8   /*  receiver begin of break */
#define UTSR0_REB   16  /*  receiver end of break */
#define UTSR0_EIF   32  /*  error in fifo */
```

3.9 Extended API initialization functions

The entry point to an ARM program is defined by either the `-entry` option of armlink or the assembler `ENTRY` directive. µHAL attaches this directive to the routine `__main` in `sources\boot.s`. µHAL places the exception vectors at the start of the image so that the application functions correctly from RAM or ROM at address 0. When used in a system with static memory at address 0, the default vectors in sources\`boot.s` must be changed to correspond to the ones actually used in the high-level application.

All ARM processors execute their first instruction at address 0. In many systems, however, this address contains volatile RAM. Implementations that assert the **HIVECS** input pin to start CPU execution from an address other than 0 are not supported by µHAL. Each system must implement a mechanism to allow static memory, such as flash or ROM, to overlay this RAM so the program can start.

The startup procedure is:

1. Switch the memory map back to its normal layout by using the `GOTO_ROM` macro in the target specific `target.s` file.
2. Initialize the memory systems (if necessary) and determine the amount of RAM in the system.
3. If the application is compiled standalone, copy the exception vectors from static memory to RAM, starting at address 0.
4. Set up the stacks for the different processor modes and initialize the predefined data areas for the high-level application.
5. Initialize the rest of the system, including MMU, cache, serial ports, interrupts, and timers.

Some applications might hide this completely within the boot-up section. Others set up only the required functions from within the application.

The initialization functions are:

- *uHALir_InitTargetMem()* on page 3-33
- *uHALir_InitBSSMemory()* on page 3-33
- *uHALir_PlatformInit()* on page 3-33.

3.9.1 uHALir_InitTargetMem()

This function checks and initializes the memory system and then returns the address of the top of available memory. This function does not corrupt memory if it is already initialized.

Note

This routine cannot be called from C because it assumes there is no stack and that registers do not have to be preserved.

Syntax

```
void *uHALir_InitTargetMem(void)
```

Return value

Returns `Top of Memory +1` (in bytes) and stores it in `uHALiv_TopOfMemory`.

3.9.2 uHALir_InitBSSMemory()

This function initializes the predefined variables and zeroed memory used by the high-level application.

Note

This routine overwrites any variables declared within the application.

Syntax

```
void uHALir_InitBSSMemory(void)
```

3.9.3 uHALir_PlatformInit()

This function initializes any platform-specific systems that must be setup before control is passed to the application.

Syntax

```
void uHALir_PlatformInit(void)
```

3.10 Extended API interrupt handling functions

The ARM processor has IRQ and FIQ interrupts. µHAL avoids using FIQs, leaving them available to the debugger and/or user application.

How µHAL initializes interrupts depends on the mode you have built it to execute in:

- For standalone applications, the full set of vectors (contained in `\sources\boot.s`) is usually copied to memory at physical address `0x00000000`.
- For semihosted applications, µHAL installs an interrupt handler when the application requests one. This vector contains the address of `uHALir_IRQProcess()`, a dummy IRQ handler.

When IRQs are installed (using `uHALr_InitInterrupts()`), µHAL installs a pointer to the default trap handling function `uHALr_TrapIRQ()` (in `Sources\irqtrap.s`) into the exception vector at offset `0x18`.

When an interrupt occurs, `uHALr_TrapIRQ()` saves all the registers in an APCS-compliant manner and, optionally, calls several handler routines to actually handle the IRQ. These handlers are called:

- after the context has been saved on the IRQ stack
- after the source of the interrupt has been determined
- at the end of the interrupt, just before the PC is restored from LR.

These routine addresses are stored in `uHALp_StartIRQ`, `uHALp_HandleIRQ`, and `uHALp_FinishIRQ`, respectively. The interrupt exception vector is modified using `uHALir_NewVector()`.

The interrupt handler functions are:

- *uHALir_TrapIRQ()* on page 3-35
- *uHALir_NewVector()* on page 3-35
- *uHALir_NewIRQ()* on page 3-36
- *uHALir_DefineIRQ()* on page 3-36
- *uHALir_DispatchIRQ()* on page 3-37
- *uHALir_UnexpectedIRQ()* on page 3-38.

3.10.1 uHALir_TrapIRQ()

This function:

1. saves all the registers in an APCS-compliant manner
2. calls a `StartIRQ()` function, if defined
3. reads the interrupt mask and calls the high-level handler `HandleIRQ()`
4. calls a `FinishIRQ()` function, if defined
5. jumps to a returned value as an address to finish IRQ processing, if `FinishIRQ()` returns this value.

You can specify your own handler to use instead of the default trap handler by directly calling this low-level interrupt installer (found in `Sources\irqlib.s`). The application must completely handle its own interrupts. µHAL itself calls this routine from `uHALr_InitIRQ()` to install the low-level trap handler `uHALr_TrapIRQ()` and the high-level IRQ dispatcher `uHALr_DispatchIRQ()`.

Note

This function is intended as an IRQ handler and not a user-called function.

Syntax

```
void uHALir_TrapIRQ(void)
```

3.10.2 uHALir_NewVector()

This function replaces the specified exception vector with the given routine pointer.

Note

This routine is not ACPS-compliant as it may be called before stacks and memory are defined. The function must be called in Supervisor mode.

Syntax

```
int *uHALir_NewVector(void *Vector, PrVoid LowLevel)
```

where:

Vector is the address of the vector to be replaced.

LowLevel is a pointer to the low-level exception handler.

Register contents

The register contents on return are:

r2 The address of the old vector, branch, or NULL.

r1 The address of the new vector, branch, or NULL.

r0 The status:

- 0 if the new exception vector could not be written
- 1 if the old exception was an LDR PC instruction
- 2 otherwise.

3.10.3 uHALir_NewIRQ()

This function installs both the high-level and low-level IRQ routines. To install the low-level routine, its address is copied to the vector array used by the exception vectors.

Assuming that the application chooses to use the default trap handler, µHAL allows the application to specify handlers for the start and end of each interrupt, in addition to allowing it to actually handle the interrupt. For simple interrupts, only the IRQ handler is needed. However, some operating systems ported to µHAL make use of the start and finish handlers to aid context switching (typically done at the end of timer interrupt handling). Use this function to define any of these three interrupt handlers.

Syntax

```
void uHALir_NewIRQ(PrHandler HighLevel, PrVoid LowLevel)
```

where:

HighLevel is a pointer to the high-level routine that processes interrupts. By default, this is `uHALr_DispatchIRQ()`.

LowLevel is a pointer to the low-level routine. This routine must switch out of IRQ mode and restore correct operation of the application upon completion. If this pointer is zero, the default routine `uHALr_TrapIRQ()` is installed.

3.10.4 uHALir_DefineIRQ()

This function allows some or all of the functionality of the low-level IRQ handler to be defined. If zero is passed as the pointer contents, no action is taken for that parameter.

This is the default interrupt dispatcher (found in `Sources\irq.c`). This is passed the interrupt sources as a 32-bit value. How the interrupt sources are determined is board-specific and the `READ_INT` macro in `target.s` is used for this purpose.

Syntax

```
void uHALir_DefineIRQ(PrVoid Start, PrPrVoid Finish,
                      PrVoid Trap)
```

where:

Start is a pointer to the routine to be executed at the start of every IRQ.

Finish is a pointer to the routine to be executed at the finish of every IRQ.

Trap is a pointer to a different low-level IRQ handler. This handler might function differently than the default operation. The default interrupt routine is `uHALr_TrapIRQ()`.

Usage

Start and *Finish* are zero if not required, but if *Trap* is zero, the current vector is not overwritten. This routine should be called before the call to `uHALr_InitInterrupt()`.

3.10.5 uHALir_DispatchIRQ()

This is the high-level interrupt handler that scans the IRQ flags to find the interrupt that caused the exception. The appropriate installed interrupt handler is then called. If no handler is found, a common unexpected IRQ routine is called.

The interrupts themselves are owned by an interrupt handler. The µHAL code in `Sources\irq.c` maintains the `uHALv_IRQVector` array of `uHALis_IRQ` structs that describe the handler for each interrupt source.

The format of the data structure is:

```
struct uHALis_IRQ {
   PrHandler  handler ;     /* Routine for */
                            /* specific interrupt */
   unsigned int   flags ;
   unsigned int   mask ;
   const  unsigned char *name ; /* Debug, owner id */
   struct uHALis_IRQ *next ;  /* Useful for shared */
                              /* interrupts */
};
```

There are `NR_IRQS` elements. `NR_IRQS` is defined in the board-specific `platform.s` and `platform.h` files. µHAL applications install interrupt handlers using the `uHALr_RequestInterrupt()` function described on page 3-9. The application enables the interrupt by calling `uHALr_EnableInterrupt()`.

This function unmasks this particular interrupt by calling the board-specific `uHALir_UnmaskIrq()` function (found in `board.c`), and enables IRQs in the processor by calling `uHALir_EnableInt()` (found in `Sources\lib\irqlib.s`).

When the interrupt occurs, `uHALr_DispatchIRQ()` calls the interrupt handler for every pending bit set in the interrupt flags. To enable an application to have one interrupt handler for several interrupts, the interrupt number is passed to the interrupt handler. If an interrupt occurs and there is no installed interrupt handler, the unexpected interrupt handler is called (see *uHALir_UnexpectedIRQ()* on page 3-38).

Syntax

```
void uHALir_DispatchIRQ(unsigned int irqflags)
```

where:

irqflags is the pending interrupt or interrupts.

Note

This function is intended as an IRQ handler and not a user-called function.

3.10.6 uHALir_UnexpectedIRQ()

This function prints a debug message and some status information when an interrupt is received for which no handler has been installed.

The function can be adapted to disable the interrupt by adding a call to `uHALr_DisableIRQ()`.

Syntax

```
void uHALir_UnexpectedIRQ(unsigned int irq)
```

where:

irq is the number of the interrupt that triggered unexpectedly.

3.11 Extended API software interrupt (SWI) function

A *Software Interrupt Instruction* (SWI) provides a means for a program running in User mode to request privileged operations that must be run in Supervisor mode. The only SWI currently handled by µHAL is `SWI_EnterOS`. This SWI switches into Supervisor mode. All other SWIs, and the behavior of µHAL with these SWIs, are undefined.

Note

When running µHAL under a debug agent, such as Angel, the SWI exception vector is not overwritten. It is the debug agent that executes the SWI handler. Also, character input/output is handled by the debug agent rather than being directly sent to or received from the serial port.

3.11.1 uHALir_TrapSWI()

This function handles SWI exceptions. The only SWI currently decoded is `SWI_EnterOS`. This SWI returns back to the initial context in Supervisor mode.

Note

Because the SWI call writes the return address into the link register (written as lr or r14), the link register must be protected. This is part of the µHAL support code, and it is not intended to be called by user programs.

Syntax

```
void uHALir_TrapSWI(void)
```

3.12 Extended API MMU and cache functions

Memory management is a complex issue. Refer to the *ARM Architecture Reference Manual* for more details.

µHAL provides two basic routines to reset the MMU to its power-on state (that is, disabled) and to initialize a one-to-one mapping, as described in *Simple API MMU and cache functions* on page 3-11.

Safe, finer control of the MMU is also provided by the processor-specific functions described below. The common shell of these functions is contained in the `mmu.s` or `cache.c` modules (for example, `Processors\mmu.s`). The unique features of each processor are implemented using macros (for example, `Processors\ARM720T\mmu720T.s`). The functions are designed to operate safely, stay in SVC mode, and maintain cache coherency. In order to correctly clean a cache, the code might require board-specific information on which addresses it can use.

The cache management functions are:

- *uHALir_EnableICache()*
- *uHALir_DisableICache()* on page 3-41
- *uHALir_EnableDCache()* on page 3-41
- *uHALir_DisableDCache()* on page 3-41
- *uHALir_CleanDCache()* on page 3-41
- *uHALir_CleanDCacheEntry()* on page 3-42
- *uHALir_EnableWriteBuffer()* on page 3-42
- *uHALir_DisableWriteBuffer()* on page 3-42
- *uHALir_ReadCacheMode()* on page 3-42
- *uHALir_WriteCacheMode()* on page 3-43.

3.12.1 uHALir_EnableICache()

This function enables the instruction cache only. If the processor does not support a separate instruction cache, the cache is enabled for both instructions and data. If the processor has no caches, no action is taken.

Syntax

```
void uHALir_EnableICache(void)
```

3.12.2 uHALir_DisableICache()

This function disables the instruction cache only. If the processor has a combined data and instruction cache, then it is disabled. If the processor has no caches, no action is taken.

Syntax

```
void uHALir_DisableICache(void)
```

3.12.3 uHALir_EnableDCache()

This function enables the data cache only. If the processor has a combined data and instruction cache, then it is enabled. If the processor has no caches, no action is taken.

Syntax

```
void uHALir_EnableDCache(void)
```

3.12.4 uHALir_DisableDCache()

This function disables the data cache only. If the processor has a combined data and instruction cache, then it is disabled. If the processor has no caches, no action is taken.

Syntax

```
void uHALir_DisableDCache(void)
```

3.12.5 uHALir_CleanDCache()

This function cleans the data cache. If the processor has a combined data and instruction cache, then it is cleaned. If the processor has no caches, no action is taken.

Syntax

```
void uHALir_CleanDCache(void)
```

3.12.6 uHALir_CleanDCacheEntry()

This function cleans the data cache entry for the specified address. If the processor does not support cleaning of individual data cache entries, the whole data cache is cleaned. If the processor does not support a separate data cache, the combined data and instruction cache is cleaned. If the processor has no caches, no action is taken.

Syntax

```
void uHALir_CleanDCacheEntry(void *address)
```

where:

address Is the location to be synchronized with memory.

3.12.7 uHALir_EnableWriteBuffer()

This function enables the write buffer. If the processor does not support a write buffer, the operation is zero.

Syntax

```
void uHALir_EnableWriteBuffer(void)
```

3.12.8 uHALir_DisableWriteBuffer()

This function disables the write buffer. If the processor does not support a write buffer, no action is taken.

Syntax

```
void uHALir_DisableWriteBuffer(void)
```

3.12.9 uHALir_ReadCacheMode()

This function reads the current MMU and cache modes.

Syntax

```
unsigned int uHALir_ReadCacheMode(void)
```

Return value

Returns the current mode of the cache and MMU.

3.12.10 uHALir_WriteCacheMode()

This function updates the processor MMU and cache state.

Syntax

```
void uHALir_WriteCacheMode(unsigned int mode)
```

where:

mode is any combination of the MMU mode flags and cache bit flags:

`EnableMMU`
: Enables the memory management unit.

`IC_ON`
: Turns the I cache on.

`DC_ON`
: Turns the D cache on.

`WB_ON`
: Turns the Write Buffer on.

Example

The following code enables all caching:

```
void uHALr_EnableCache(void)
{
  intmode = uHALir_ReadCacheMode() ;
  uHALir_WriteCacheMode(mode | (IC_ON + DC_ON + WB_ON)) ;
}
```

3.13 Extended API processor execution mode functions

The ARM architecture (version 4 and later) has seven processor modes (as described in the *ARM Architecture Reference Manual*). Supervisor, System, and User modes apply to normal program execution.

These modes differ in priority, access to registers, memory, and peripherals. System and User modes use the same stack and registers. Application code often executes in the non-privileged User mode and cannot directly change the interrupt bits in the CPSR.

The processor execution mode functions (in `Sources\cpumode.s`) allow the current mode to be read and changed. When switching processor mode, the application must protect the original *Saved Program Status Register* (SPSR), especially when in an interrupt, so that functionality can be fully unwound.

The processor execution mode functions are:

- *uHALir_EnterSvcMode()*
- *uHALir_ExitSvcMode()* on page 3-45
- *uHALir_EnterLockedSvcMode()* on page 3-45
- *uHALir_ReadMode()* on page 3-45
- *uHALir_WriteMode()* on page 3-46.

3.13.1 uHALir_EnterSvcMode()

This function switches the mode to Supervisor mode, irrespective of the current mode. It masks some considerations regarding `SWI_EnterOS`. The calling routine must save the returned value to be passed to `uHALir_ExitSVCMode()`.

Note

You must take care to balance stacks according to processor mode.

Syntax

```
unsigned int uHALir_EnterSvcMode(void)
```

Return value

Returns SPSR, the initial saved processor mode (used to restore the mode by `uHALir_ExitSVCMode()`).

3.13.2 uHALir_ExitSvcMode()

This function restores the mode back to the original mode. It switches back to the original processor mode before the call to `uHALir_EnterSVCMode()`, and restores the original SPSR, as saved by the calling routine.

Syntax

```
void uHALir_ExitSvcMode(unsigned int spsr)
```

where:

spsr is the original SPSR.

3.13.3 uHALir_EnterLockedSvcMode()

This function switches into Supervisor mode and disables IRQ interrupts. It masks some of the considerations regarding `SWI_EnterOS`,

Note

You must take care to balance stacks according to processor mode.

Syntax

```
unsigned int uHALir_EnterLockedSvcMode(void)
```

Return value

Returns the original SPSR.

3.13.4 uHALir_ReadMode()

This function reads the current execution mode.

Syntax

```
unsigned int uHALir_ReadMode(void)
```

Return value

Returns the *Current Program Status Register* (CPSR).

3.13.5 uHALir_WriteMode()

This function changes the current execution mode.

Note

The processor must already be in a privileged mode (not in User mode).

Syntax

```
void uHALir_WriteMode(unsigned int cpsr)
```

where:

cpsr is the new CPSR.

3.14 Extended API timer functions

The timer code is held in the module `Sources\timer.c`. It stores information about the timers in the system in the `uHALiv_TimerStatus` vector. This is an array of `uHALis_Timer` data structures that must have the following format:

```
/* Enum to describe timer: free, one-shot, on-going interval */
/* or locked-out */
enum uHALe_TimerState
                  { T_FREE, T_ONESHOT, T_INTERVAL, T_LOCKED } ;

struct uHALis_Timer {
    unsigned int irq ;            /* IRQ number */
    enum uHALe_TimerState state ;
    unsigned int period ;         /* Period between triggers */

    PrHandler handler;            /* User Routine */
    const unsigned char *name;    /* Debug, owner id */
    PrHandler ClearInterruptRtn;  /* User Routine */
    int hw_interval:1;
    struct uHALis_Timer *next;
} ;
```

µHAL also maintains a second vector, `uHALiv_TimerVectors`. This contains the interrupt number for each timer. `uHALiv_TimerVectors` is initialized to the value of `TIMER_VECTORS`. This, along with `MAX_TIMER`, `HOST_TIMER` and `OS_TIMER`, is defined in the board-specific `platform.h` and `platform.s` files.

`MAX_TIMER` is the number of timers in the system. `HOST_TIMER` is the timer being used by the debug agent (for example, Angel), and `OS_TIMER` is the timer that supports the system timer.

When the timer subsystem is initialized by `uHALr_InitTimers()`, it sets up the contents of the `uHALiv_TimerStatus` vector. If there is a `HOST_TIMER` defined, that timer state becomes `T_LOCKED`. Otherwise, it is set to `T_FREE`. By default, the timer length is set to one millisecond. This value is also defined in `platform.h` and `platform.s` (as the literal `mSEC_1`). At initialization time, a free timer is disabled by calling the board-specific `uHALir_PlatformDisableTimer()` function in `board.c`.

After the application uses a `uHALr_RequestTimer()` call to assign a particular timer, it can read and alter the timer state and interval. However, the application cannot alter the maximum length of time that a timer can run for. This is defined by `MAX_PERIOD`.

µHAL timers depend on the µHAL interrupt handling system to operate. When a timer is initialized by the application calling `uHALr_InstallTimer()`, the µHAL timer subsystem assigns the IRQ to the timer interrupt handler (`uHALir_TimeHandler()`) and enables the timer so that it can start running. The enabling of the timer is done by the board-specific `uHALir_PlatformEnableTimer()` function in `board.c`.

The timer handler is responsible for handling the timer interrupts as they occur:

1. The handler is passed the IRQ of the interrupting timer, and uses this to determine the timer that has expired.
2. After the handler has discovered which timer has expired, it calls the handler function for that timer.
3. If the timer was a single-shot timer (its state is `T_ONESHOT`), the timer is automatically freed by calling `uHALr_FreeTimer()`. Otherwise, the timer is left to run.

The timer functions are:

- *uHALir_TimeHandler()*
- *uHALir_DisableTimer()* on page 3-49
- *uHALir_GetTimerInterrupt()* on page 3-49.

3.14.1 uHALir_TimeHandler()

This is a high-level function that:

1. determines which timer caused the interrupt
2. calls its handler
3. determines if the timer should be cancelled or re-enabled.

Syntax

```
void uHALir_TimeHandler(unsigned int irqflags)
```

where:

irqflags is the currently pending interrupt.

Note

This is not a user-callable function.

3.14.2 uHALir_DisableTimer()

This function disables the specified timer.

An application typically frees the timer when it has finished with it. µHAL also allows the timer to be disabled. In this case, the timer subsystem calls the platform-specific `uHALir_PlatformDisableTimer()` function (found in `board.c`) to do the work.

Syntax

```
void uHALir_DisableTimer(unsigned int timer)
```

where:

`timer` is the timer to be disabled.

3.14.3 uHALir_GetTimerInterrupt()

This function allows the application to determine the correct interrupt for the specified timer. Different target systems may assign different interrupts to the timer.

Syntax

```
int uHALir_GetTimerInterrupt(unsigned int timer)
```

where:

`timer` is the timer number for which the interval is requested.

Return value

Returns one of the following:

interrupt If the timer is found, the interrupt number is returned.

-1 If the timer is not found.

3.14.4 uHALir_GetSystemTimer()

This function returns the timer number defined as the system timer.

Syntax

```
unsigned int uHALir_GetSystemTimer(void)
```

Return value

Returns the number of the IRQ for the system timer.

3.15 Extended API coprocessor access functions

These routines allow access to some of the registers in the MMU coprocessor. This allows the processor ID to be read, and the MMU/cache configuration to be read and modified.

The coprocessor access functions are:

- *uHALir_CpuIdRead()*
- *uHALir_CpuControlRead()*
- *uHALir_CpuControlWrite()* on page 3-52.

3.15.1 uHALir_CpuIdRead()

This function reads the processor ID. Reading from CP15, r0 returns an architecture and implementation-defined identification from the processor. If there is no cache, MMU, or write buffer, this routine returns a value equivalent to ARM7.

Syntax

```
unsigned int uHALir_CpuIdRead(void)
```

Return value

Returns the CPU type as read from the register.

3.15.2 uHALir_CpuControlRead()

This function reads from the appropriate coprocessor register to read the current state of the MMU and caches. If there is no cache, MMU, or write buffer, this routine returns 0 (all disabled).

Syntax

```
unsigned int uHALir_CpuControlRead(void)
```

Return value

Returns the MMU/cache control state as read from the register.

3.15.3 uHALir_CpuControlWrite()

This function writes to the appropriate coprocessor register to set the state of the MMU and caches. If there is no cache, MMU, or write buffer, this routine has no effect.

Syntax

```
void uHALir_CpuControlWrite(unsigned int controlState)
```

where:

controlState

is the desired implementation-specific value for this register.

3.16 Library support functions

You can write µHAL applications that run on multiple platforms when linked with the appropriate libraries. These routines allow the application to initialize the library and to determine whether the system supports:

- PCI
- MMU or MPU
- cache
- unified or separate data and instruction caches.

The library functions are:

- *uHALr_LibraryInit()*
- *uHALir_MMUSupported()* on page 3-54
- *uHALir_MPUSupported()* on page 3-54
- *uHALir_CacheSupported()* on page 3-54
- *uHALir_CheckUnifiedCache()* on page 3-55.

Note

For information about the PCI library query function `uHALr_PCIHost ()`, see *µHAL PCI function descriptions* on page 9-16.

3.16.1 uHALr_LibraryInit()

This function performs system-specific initialization of µHAL when an application is linked to another library, such as the ADS C runtime library.

Syntax

```
void uHALr_LibraryInit(void)
```

3.16.2 uHALir_MMUSupported()

This function tests the µHAL library for MMU support.

Syntax

```
int uHALir_MMUSupported(void)
```

Return value

Returns one of the following:

1 If the library has been built for MMU access.

0 If the library has no MMU support.

3.16.3 uHALir_MPUSupported()

This function tests the µHAL library for MPU support.

Syntax

```
int uHALir_MPUSupported(void)
```

Return value

Returns one of the following:

1 If the library has been built for MPU access.

0 If the library has no MPU support.

3.16.4 uHALir_CacheSupported()

This function tests the µHAL library for cache support

Syntax

```
int uHALir_CacheSupported(void)
```

Return value

Returns one of the following:

1 If the library has been built for cache access.

0 If the library has no cache support.

3.16.5 uHALir_CheckUnifiedCache()

This function tests the µHAL library for unified cache support.

Syntax

```
int uHALir_CheckUnifiedCache(void)
```

Return value

Returns one of the following:

1 If the library has been built for unified cache access.

0 If the library has separate data and instruction cache support.

Chapter 4
ARM Boot Monitor

This chapter describes the boot monitor supplied with AFS. It contains the following sections:

See Chapter 7 *Flash Library Specification* and Chapter 8 *Using the ARM Flash Utilites* for additional information about images in flash memory.

4.1 About the boot monitor

The boot monitor is a ROM-based monitor that 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.

The boot monitor is a µHAL application. It uses the µHAL library to initialize the system when it runs.

This chapter describes how the boot monitor works and how to port the boot monitor to another hardware platform.

4.1.1 Hardware accesses

The boot monitor accesses hardware using library calls to µHAL and other firmware. This makes it generic and easily portable to platforms that support µHAL.

The boot monitor uses the following firmware libraries:

µHAL For memory initialization, heap, serial interface, timers and LEDs.

PCI For systems that support PCI, such as the Integrator, the boot monitor makes use of the PCI library.

Flash For programming images into flash and when using *System Information Blocks* (SIB).

4.1.2 Setting up a serial connection

To communicate with the boot monitor on the development board, you require a terminal emulator that can send raw ASCII data files (for example, Windows HyperTerminal). Connect a null modem cable to the serial port on the development board. (If your development board has two serial ports, refer to the hardware manual to identify the one used with boot monitor.) The terminal emulator must be set up with the following settings:

Baud rate 38400

Data bits 8

Parity none

Stop bits 1

Flow control Xon/Xoff

The ARM development boards have switches that select whether the boot monitor or an image in flash memory is started on reset. Refer to the hardware manuals for your board to identify the switch settings that enable the boot monitor.

4.1.3 Boot monitor functions

The boot monitor supplies a base set of functions that are common across all boards. These functions:

- download images using the serial line into system memory or flash memory
- read and display words in memory
- erase system flash memory
- use the µHAL library to test all of the features available
- identify the board (including hardware and software revisions).

Board-specific extensions to the boot monitor

The boot monitor allows this functionality to be extended with board-specific commands and self-tests. For information on the boot monitor commands specific to the Integrator board, see *Integrator-specific commands for boot monitor* on page 4-12. For information on the boot monitor commands specific to the Prospector board, see *Prospector-specific commands for boot monitor* on page 4-22.

4.2 Common commands for the boot monitor

The command interpreter accepts user commands and carries out actions to complete the commands. Table 4-1 lists the basic commands for the boot monitor. The commands required as part of the Microsoft HARP/SDB specification are marked *Yes*.

Table 4-1 Boot monitor commands

Command	Required	Action
`B` *`number`*	Yes	Set the baud rate for the serial line to *`number`*.
`BI` *`number`*	No	Make image *`number`* the image to boot when S1 is OFF.
`D` *`address`*	Yes	Read and display eight 32-bit words starting from *`address`*. (Specify *`address`* in hex format.)
`E`	Yes	Erase all of the application flash and return the prompt when complete.
`H` or `?`	No	Display help.
`I`	Yes	Print out board information, including identifying the board, its hardware, and software revision.
`L`	Yes	Run the Motorola S-record loader. Subsequent serial data is interpreted in the standard S-record format and written to flash.
`M`	No	Download an image into RAM. Subsequent data is interpreted as S-record format.
`T`	Yes	Run system self tests.
`V`	No	Validate flash, including the system information blocks.
`X` *`command`*	No	Enter board-specific command mode and execute *`command`*.

Note

Commands are accepted in upper or lower case.

There are additional, or modified, commands specific for the Integrator and Prospector boards. Table 4-2 on page 4-12 lists the commands for Integrator and Table 4-4 on page 4-22 lists the commands for Prospector. For more information on using the boot monitor with Integrator, see *Using the boot monitor on Integrator* on page 4-25.

4.2.1 B: Set baud rate

This command is used to set the baud rate for the serial line used by the boot monitor. For example:

```
boot Monitor > b 115400
```

The baud rate changes immediately after the reply is sent. You must reconfigure your terminal emulator to use the new baud rate in order to send new commands.

The flow control and stop bits are not reconfigurable. See *Setting up a serial connection* on page 4-2 for other serial port settings.

Note

If the boot monitor is used on a system that requires the MMU to be active, such as the Prospector P1100, any attempt to read from or write to an invalid address causes a data exception and the boot monitor resets.

4.2.2 BI: Set default flash boot image number

This command sets the default flash boot image to the image number specified. This modifies the boot monitor SIB. Entering the command without specifying an image number returns the number of the currently selected boot image. The image number is the logical image number, and is not based on the order of the images in flash.

Use the ARM Flash Utility to load multiple images into flash. See *AFU commands* on page 8-4.

Examples of this command are shown in Example 4-1.

Example 4-1 Set default boot image

```
boot Monitor > bi
Current Boot Image = 0
boot Monitor > bi 1
Current Boot Image = 0
New Boot Image = 1
```

4.2.3 D: Display system memory

This command displays eight 32-bit words of system memory at the address given. An example of this command is shown in Example 4-2.

Example 4-2 Display system memory

```
boot Monitor > d 0x24000000
Displaying memory at 0x24000000
0x24000000: 0xE59FF018
0x24000004: 0xE59FF018
0x24000008: 0xE59FF018
0x2400000C: 0xE59FF018
0x24000010: 0xE59FF018
0x24000014: 0xE59FF018
0x24000018: 0xE59FF018
0x2400001C: 0xE59FF018
```

4.2.4 E: Erase application flash

This command erases all of the application flash, including all of the SIBs. You are prompted to confirm that you want to proceed or cancel the command. After the flash has been erased, the boot monitor SIB is recreated. The boot monitor SIB is changed to run image number zero on reset. An example is shown in Example 4-3.

Example 4-3 Erase system flash

```
boot Monitor > e
Erase all of the system flash
Are you sure that you want to do this[Ny]? y
Erasing all flash

.............................
.............................
.............................
.............................
.............................
.............................
.............................
.............................

Initializing Boot Monitor System Information Block
```

4.2.5 H or ?: Display help

This command lists the full set of commands for this mode, as listed in Table 4-1 on page 4-4.

4.2.6 I: Identify the system

This command identifies the system on which the boot monitor is installed. It prints a message similar to that shown in Example 4-4.

Example 4-4 Identify the system

```
boot Monitor > i
ARM bootPROM [Version 1.0] Rebuilt on May 20 1999 at 12:24:07
Running on a Integrator (Board revision v1.0, ARM720T Processor)
Memory Size is 0x2000000 bytes, Flash size is 0x2000000 bytes
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
```

4.2.7 L: Load S-records into flash

This command downloads an image into memory and then programs it into flash. As part of the programming process it builds an appropriate flash image footer.

By default, the image is written to the location specified by the address in the S-records and labeled as image number 0. This might overwrite one or more images. When the image has been successfully written into flash, the boot monitor SIB is updated so that the default image to boot from flash is image 0.

The downloaded image is given the name `BMON Loaded`. The original image number 0 and any images wholly or partially overwritten are deleted.

To load a file:

1. Type `l` at the prompt.
2. Use the **Transmit File** command of your terminal emulator to send the file. If the emulator has two file transfer options, use the **Send ASCII File** option.

Example 4-5 on page 4-8 shows an example of the load S-records into flash command.

The boot monitor gives an approximate progress indication by displaying a dot for every 64 received records. If your terminal emulator does not give progress indication as the file downloads, use the displayed dots as a guide and wait a sufficient time for the file to download. Press `Ctrl+C` after the file has finished loading to prompt the boot monitor to terminate the download and display the number of records downloaded.

The name `BMON Loaded` is given to any image that is loaded by the boot monitor.

As with all serial commands, the terminal emulator must use Xon/Xoff flow control. If you do not have Xon/Xoff flow control enabled, the boot monitor may seem to work correctly for commands that do not require a large number of bytes to be exchanged, but then might not work reliably when large files are loaded.

Example 4-5 Load S-records into flash

```
boot Monitor > l
Load Motorola S-Records into flash
Deleting Image 0

Type Ctrl/C to exit loader.

.......................................
........................


Downloaded 697 records in 10 seconds.
Overwritten block/s
     0
boot Monitor >
```

4.2.8 M: Download an image into RAM

Use this command to download an image into RAM at addresses specified in the S-records (the addresses must be valid memory addresses). Once the image has been downloaded, control of the system is transferred to that image. An example is shown in Example 4-6.

To load a file:

1. Type `m` at the prompt.
2. Use the **Transmit File** command of your terminal emulator to send the file. If the emulator has two file transfer options, use the **Send ASCII File** option.
3. Enter `Ctrl+C` to indicate to the boot monitor that the image has been loaded.

Example 4-6 Download image

```
Load Motorola S-Record image into memory and execute it
Record addresses must be between 0x00008000 and 0x01FD9DFF.
Type Ctrl/C to exit loader.
```

4.2.9 T: System self tests

This command starts the system self tests. The system self tests are used to check that the system is functioning correctly. They make use of the resources that are known to function reliably (these resources vary between platforms but must include one UART).

The tests include:

- counter/timer checks
- LED checks.

You can extend these tests by board specific tests. shows an example of a default self test.

System self tests

```
boot Monitor > t
Generic Tests
Type any character to abort the tests
Initializing self test environment
Timer tests
  Running Timer tests
  ++++++++++
  Timer tests successful
LED flashing test
  Lighting all 4 LEDs in sequence
Did you see the LEDs flash in sequence[Yn]? y
...performed 2 tests, 0 failures
Board Specific Tests
Type any character to abort the tests
Keyboard/mouse tests

Initializing KMI interface
==========================

kmi_handler(3)
kmi_handler(3)
KMI: wrote FF
kmi_handler(4)
kmi_handler(4)
KMI: wrote FF
    Port 0: Device unsupported or absent
    Port 1: Device unsupported or absent

...performed 1 tests, 0 failures
```

4.2.10 V: Validate flash

This command validates and displays the contents of the application flash and the SIBs. It flags any errors that it finds. An example is shown in Example 4-7. The name `BMON Loaded` is given to any image that is loaded by the boot monitor.

Example 4-7 Validate flash

```
boot Monitor > v
There are 256 128Kbyte blocks of flash:

Images found in flash
=========================
Block  Size  ImageNo  Name
-----  ----  -------  ----
    0     1        0  BMON Loaded

System Information Blocks
=========================
Owner                Size
-----                ----
ARM Boot Monitor      260
boot Monitor >
```

4.2.11 X: Enter board-specific command mode

This mode is used to process board-specific (or extended) commands. If you enter a single `X`, the prompt changes to show that you are in the extended mode. The board-specific menu provides a command that returns you to the normal command processing mode. To exit board-specific mode, enter an `X` in extended mode.

You can execute a single board-specific command by entering a command on the same line as X (with a space in between).

4.3 Integrator-specific commands for boot monitor

The Integrator provides a set of system specific boot monitor commands. These are listed in Table 4-2. Examples are provided in *I: Initialize or re-initialize the PCI sub-system* on page 4-13 to *H or ?: Display help* on page 4-21.

Table 4-2 Integrator system-specific commands

Command	Action
I	Initialize or re-initialize the PCI subsystem
V	Display V3 chip setup
P	Display PCI topology
DPI *hex*	Display PCI IO space (32 bit reads)
DPM *hex*	Display PCI Memory space (32 bit reads)
DPC *hex*	Display PCI Configuration space (32 bit reads)
CC	Set clocks from SIB
DC	Display clock frequencies
SCC *number*	Set core clock frequency in SIB (MHz)
SMC *number*	Set memory bus clock frequency in SIB (MHz)
SSC *number*	Set system bus clock frequency in SIB (MHz)
SPC *number*	Set PCI clock frequency in SIB (MHz)
DH	Display hardware
G *hex*	Go to address
X	Exit board specific command mode
X *command*	Execute single non board-specific command
?	Display help
H:	Display help

4.3.1 I: Initialize or re-initialize the PCI sub-system

This command causes the PCI subsystem to be re-initialized. You are prompted to confirm the command before re-initialization is carried out. See Example 4-8.

Example 4-8 Initialize result

```
[Integrator] boot Monitor > i
About to re-initialise PCI
Are you sure that you want to do this[Yn]? y
Initialising PCI...done
```

4.3.2 V: Display V3 chip setup

This command displays the current set up of the V3 host bridge. See Example 4-9.

Example 4-9 Display V3 result

```
[Integrator] boot Monitor > v
V3 PCI Host Bridge (@ 0x62000000)
 [0x00000078] SYSTEM      : 0xC000(Locked, Reset output de-asserted)
 [0x0000007C] PCI_CFG      : 0x1166
 [0x0000007A] LB_CFG       : 0x00C0
 [0x00000004] PCI_CMD      : 0x0006
Local --> PCI windows:
 [0x00000054] LB_BASE0     : 0x40000081
 [0x0000005E] LB_MAP0      : 0x4006
 [0x00000058] LB_BASE1     : 0x50000081
 [0x00000062] LB_MAP1      : 0x5006
 [0x00000064] LB_BASE2     : 0x6001
 [0x00000066] LB_MAP2      : 0x0000
PCI --> Local windows:
 [0x00000010] PCI_IO_BASE  : 0x00000000
 [0x00000014] PCI_BASE0    : 0x20000000
 [0x00000040] PCI_MAP0     : 0x20000093
 [0x00000018] PCI_BASE1    : 0x80000000
 [0x00000044] PCI_MAP1     : 0x800000A3
FIFOs:
 [0x00000070] FIFO_CFG     : 0x0000
 [0x00000072] FIFO_PRIORITY: 0x0000
 [0x00000074] FIFO_STAT    : 0x0505
 [0x0000002C] SUB_VENDOR   : 0x0000
 [0x0000002E] SUB_ID       : 0x0000
```

4.3.3 P: Display PCI topology

This command displays the topology of the PCI subsystem. It lists the devices found and their locations in PCI address space. See Example 4-10.

Example 4-10 PCI topology result

```
[Integrator] boot Monitor > p
Bus Slot Func Vendor Device Rev      Class           Cmd
=== ==== ==== ====== ====== ==== ================   ====
 00  12   00  0x10EE 0x3FC2 0x00 Multimedia Audio   0x02

      Reg    Address     Type
     ====   ==========   ======
     0x10   0x40000000   Memory
     0x14   0x00000000   Memory
     0x18   0x00000000   Memory
     0x1C   0x00000000   Memory
     0x20   0x00000000   Memory
     0x24   0x00000000   Memory
     0x30   0x00000000   ROM
     0x3D   0x00000001   Interrupt Pin
     0x3C   0x00000010   Interrupt Line

Bus Slot Func Vendor Device  Rev     Class           Cmd
=== ==== ==== ====== ======  ==== ================   ====
 00  11   00  0x1011  0x0019 0x30 Ethernet           0x07

      Reg   Address      Type
     ====  ==========   ======
     0x10  0x00004000   IO
     0x14  0x41000000   Memory
     0x18  0x00000000   Memory
     0x1C  0x00000000   Memory
     0x20  0x00000000   Memory
     0x24  0x00000000   Memory
     0x30  0x41040000   ROM
     0x3D  0x00000001   Interrupt Pin
     0x3C  0x0000000F   Interrupt Line
```



```
Bus Slot Func Vendor Device Rev      Class          Cmd
=== ==== ==== ====== ====== ==== ================  ====
 00   10   00 0x5333 0x88D0 0x00 VGA Device         0x02

     Reg   Address      Type
     ====  ==========   ======
     0x10  0x41800000   Memory
     0x14  0x00000000   Memory
     0x18  0x00000000   Memory
     0x1C  0x00000000   Memory
     0x20  0x00000000   Memory
     0x24  0x00000000   Memory
     0x30  0x42000000   ROM
     0x3D  0x00000000   Interrupt Pin
     0x3C  0x00000000   Interrupt Line

Bus Slot Func Vendor Device Rev      Class          Cmd
=== ==== ==== ====== ====== ==== ================  ====
 00   09   00 0x1011 0x0024 0x03 PCI->PCI Bridge    0x07

     Reg   Address      Type
     ====  ==========   ======
     0x10  0x00000000   Memory
     0x14  0x00000000   Memory
     0x18  0x00010100   Memory
     0x1C  0x02804150   IO
     0x20  0x42004210   Memory
     0x24  0x00010000   IO
     0x30  0x00000000   ROM
     0x3D  0x00000000   Interrupt Pin
     0x3C  0x00000000   Interrupt Line
[Integrator] boot Monitor >
```

4.3.4 DPI: Display PCI I/O space

This command displays contents of PCI I/O space at the address specified. Use hex notation for the address. See Example 4-11.

Example 4-11 Display result

```
[Integrator] boot Monitor > dpi 0x100
Displaying PCI IO memory at 0x100
0x00000100: 0xFFFFFFFF
0x00000104: 0xFFFFFFFF
0x00000108: 0xFFFFFFFF
0x0000010C: 0xFFFFFFFF
0x00000110: 0xFFFFFFFF
0x00000114: 0xFFFFFFFF
0x00000118: 0xFFFFFFFF
0x0000011C: 0xFFFFFFFF
```

4.3.5 DPM: Display PCI memory space

This command displays contents of PCI Memory space at the address specified. Use hex notation for the address. See Example 4-12.

Example 4-12 Display PCI memory result

```
[Integrator] boot Monitor > dpm 0x100
Displaying PCI Memory at 0x100
0x00000100: 0x00002378
0x00000104: 0x20000D5C
0x00000108: 0x20000D60
0x0000010C: 0x200016D0
0x00000110: 0x20000D70
0x00000114: 0x20000D74
0x00000118: 0x200016D0
0x0000011C: 0x20000D84
```

4.3.6 DPC: Display PCI configuration space

This command displays contents of PCI Configuration space at the address specified. Use hex notation for the address. See Example 4-13.

Example 4-13 Display PCI configuration result

```
[Integrator] boot Monitor > dpc 0x100
Displaying PCI Configuration memory at 0x100
0x00000100: 0xFFFFFFFF
0x00000104: 0xFFFFFFFF
0x00000108: 0xFFFFFFFF
0x0000010C: 0xFFFFFFFF
0x00000110: 0xFFFFFFFF
0x00000114: 0xFFFFFFFF
0x00000118: 0xFFFFFFFF
0x0000011C: 0xFFFFFFFF
```

4.3.7 CC: Set clocks from SIB

Enter the command `CC` to activate the settings from the boot monitor SIB. This copies the clock settings from the SIB into the relevant hardware registers. See Example 4-14. See also *DC: Display clock frequencies* on page 4-18.

Example 4-14 Set clocks

```
[Integrator] boot Monitor > dc
             SIB  Current
             ===  =======
Core         80MHz   50MHz
Memory Bus   40MHz   20MHz
System Bus   20MHz   20MHz
PCI Bus      33MHz   33MHz
[Integrator] boot Monitor > cc
[Integrator] boot Monitor > dc
             SIB   Current
             ===  =======
Core         80MHz  80MHz
Memory Bus   40MHz  40MHz
System Bus   20MHz  20MHz
PCI Bus      33MHz  33MHz
```

It is recommended that this command is used after changing the clock settings to ensure that they work correctly on the hardware in use. Using the `CC` command to increase the clock settings also improves the performance of the S-record loader, particularly at higher line speeds.

4.3.8 DC: Display clock frequencies

Displays the current clock settings as stored in the boot monitor SIB.

Integrator has four programmable clocks. These are as follows:

- core clock
- memory bus clock
- system bus clock
- PCI clock.

These clocks are defaulted by the hardware to the values shown in Table 4-3 on page 4-18.

Table 4-3 Default clock frequencies

Clock	Default value
Core	50MHz
Memory	20MHz
System	20MHz
PCI	33MHz

The boot monitor stores settings for the clocks in the SIB. Use the DC command to display the current settings. See Example 4-15.

Example 4-15 Display settings

```
[Integrator] boot Monitor > dc
              SIB    Current
              ===    =======
Core          50MHz  50MHz
Memory Bus    20MHz  20MHz
System Bus    20MHz  20MHz
PCI Bus       33MHz  33MHz

[Integrator] boot Monitor > scc 80
[Integrator] boot Monitor > smc 40
[Integrator] boot Monitor > dc
              SIB  Current
              ===  =======
Core          80MHz   50MHz
Memory Bus    40MHz   20MHz
System Bus    20MHz   20MHz
PCI Bus       33MHz   33MHz
```

When the boot switcher transfers control to an image in flash, these settings are read from the SIB and written into the relevant hardware register in the system controller FPGA. For more information on the SIB, see *SIB functions* on page 7-34.

When the system is reset, the boot monitor always starts running with the hardware defaults. This ensures that the command interpreter operates, even if the SIB contains incorrect values.

Enter the command `CC` to activate the settings from the SIB. This copies the clock settings from the SIB into the relevant hardware registers. See *CC: Set clocks from SIB* on page 4-17.

4.3.9 SCC: Set core clock frequency in SIB

This command sets the core clock frequency in the boot monitor SIB. See also *CC: Set clocks from SIB* on page 4-17.

```
[Integrator] boot Monitor > scc 80
```

Enter the command `CC` to activate the settings from the SIB. This copies the clock settings from the SIB into the relevant hardware registers. The settings in the SIB are also activated when the boot switcher transfers control to an image.

4.3.10 SMC: Set memory bus clock frequency in SIB

This command sets the memory bus clock frequency in the boot monitor SIB. See also *CC: Set clocks from SIB* on page 4-17.

```
[Integrator] boot Monitor > smc 40
```

Enter the command `CC` to activate the settings from the SIB. This copies the clock settings from the SIB into the relevant hardware registers. The settings in the SIB are also activated when the boot switcher transfers control to an image.

4.3.11 SSC: Set system bus clock frequency in SIB

This command sets the system bus clock frequency. The value is stored in the boot monitor SIB. See also *CC: Set clocks from SIB* on page 4-17.

```
[Integrator] boot Monitor > ssc 20
```

Enter the command `CC` to activate the settings from the SIB. This copies the clock settings from the SIB into the relevant hardware registers. The settings in the SIB are also activated when the boot switcher transfers control to an image.

4.3.12 SPC: Set PCI clock frequency in SIB

This command sets the PCI clock frequency. The value is stored in the boot monitor SIB. See also *CC: Set clocks from SIB* on page 4-17.

```
[Integrator] boot Monitor > spc 33
```

Enter the command `CC` to activate the settings from the SIB. This copies the clock settings from the SIB into the relevant hardware registers. The settings in the SIB are also activated when the boot switcher transfers control to an image.

4.3.13 DH: Display hardware

This command displays information about the hardware. Currently only the version numbers of the system controller and core module FPGAs are displayed as shown in Example 4-16.

Example 4-16 Display Hardware result

```
[Integrator] boot Monitor > dh
System Controller FPGA : V46, Rev A
Core Module FPGA       : V48, Rev A

[Integrator] boot Monitor > dh
```

4.3.14 G: Go to address

This command transfers control to the address supplied. Use hex notation for the address.

4.3.15 X: Exit board-specific command mode

Enter a single `x` to exit the board-specific command mode. Enter `x` followed by a command to execute a single command and then return to board-specific mode.

4.3.16 H or ?: Display help

This lists the full set of board-specific commands for this mode.

4.4 Prospector-specific commands for boot monitor

The Prospector P1100 provides a set of system-specific boot monitor commands. These are listed in Table 4-4. Examples are provided in *H or ?: Display help* on page 4-23 to *X: Exit board-specific command mode* on page 4-24.

Table 4-4 Prospector system-specific commands

Command	Action
H or ?	Display help
V	View images in flash
R *number*	Run image *number* from flash
D *address*	Display memory at *address* (use hex format)
P *address data*	Poke *data* at *address* (use hex format for both values)
G *address*	Go to *address* (use hex format)
X *command*	Exit board specific command mode

4.4.1 H or ?: Display help

This lists the full set of board-specific commands for this mode.

4.4.2 V: View images in flash

This command displays information on the images stored in boot and application flash memory (see Example 4-17).

Example 4-17 View output

```
[Prospector P-1100] boot Monitor > v

There are 2 256KByte blocks of Boot Flash:
Images found
============
Block Size  ImageNo    Name
----- ----  -------    ----
   0   1    4,280,910  bootPROM   (0x04000000-0x0403FFEC)
   1   1          911  Angel      (0x04040000-0x0407FFEC)

There are 126 256KByte blocks of Application Flash:
Images found
============
Block Size ImageNo  Name
----- ---- -------  ----
   0     1       1  Bubble           (0x04080000-0x040BFFEC)
  62     5      62  Pics             (0x05000000-0x0513FFEC)
 110     4     110  TopCat           (0x05C00000-0x05CFFFEC)

System Information Blocks
=========================
Block Owner            Index    Size
----- -----            -----    ----
  125 ARM Boot Monitor     0    260  (0x5FC0000)
```

4.4.3 D: Display memory at address

This command displays memory at hex *address*. See Example 4-18.

Example 4-18 Display memory

```
[Prospector P-1100] boot Monitor > d 0x01000000
Displaying memory at 0x1000000
0x01000000: C8000000
0x01000004: 001800C1
0x01000008: 00180101
0x0100000C: 00200000
0x01000010: 008100C0
0x01000014: 2C030500
0x01000018: 00882A90
0x0100001C: 00040D78
```

4.4.4 P: Poke memory at address

This command inserts the hex word *data* at hex *address* in memory. See Example 4-19.

Example 4-19 Poke

```
Prospector P-1100] boot Monitor > p 0x01000010 0x12345678
Poking memory at 0x1000010 with value 0x12345678
```

4.4.5 R: Run image from flash

This command transfers control to image *number* in flash. The image number is the logical image number, and is not based on the order of the images in flash.

4.4.6 G: Go to address

This command transfers control to the hex *address* supplied.

4.4.7 X: Exit board-specific command mode

Enter a single `x` to exit the board-specific command mode. Enter `x` followed by a command to execute a single command and then return to board-specific mode.

4.5 Using the boot monitor on Integrator

This section describes how to use the system-specific aspects of the boot monitor on Integrator. This includes specific boot monitor commands, boot switcher, and hardware features as they affect components of the AFS. If you are using Multi-ICE, you can get most of the functionality of the boot monitor from the *Boot Flash Utility* (`bootFU`) .

See also *Integrator-specific commands for boot monitor* on page 4-12.

4.5.1 Flash on Integrator

Integrator has two separate areas of flash designated as boot flash and application flash. Table 4-5 provides a summary of these flash areas.

Table 4-5 Flash device usage on Integrator

Device	Size	Organization	Flash Parts	Usage
Boot flash	512K	1x512K Block	Atmel AT49LV040	Boot monitor System Controller FPGA image
Application flash	32M	256x128K Blocks	Intel 28F320S3	Angel, applications and data

Application flash

The application flash is a general purpose area that can be used to store any images or data that require to be held in nonvolatile memory. The ARM Flash Library implements a simple mechanism for storing multiple images in flash. This structure enables the boot switcher to select and run the correct boot image. The ARM Flash Utility uses the flash library to program and delete images in application flash. In Table 4-5, a block is defined as the smallest area of flash that can be independently deleted. The flash library supports storing an image in either a single block or in contiguous multiple blocks.

Boot flash

The boot flash contains the default application (usually the boot monitor), boot switcher, and the FPGA image for the system controller.

Caution

This device can be reprogrammed using BootFU. However, you must take care as incorrect programming can corrupt the FPGA image and prevent the system from booting. If the system controller FPGA image is corrupted, you must reprogram the boot flash using JTAG.

In the `Build/Integrator.b` subdirectory of the `boardUtils` directory is a file called `bootPROM.mcs`. This is an Intel hex format image that includes both the boot monitor and the system controller FPGA image. If the FPGA becomes corrupted, you can use this to reprogram the boot flash over the JTAG connector.

Location of images in flash

The normal location for Angel is block 0 in the application flash. This is so that it can be run without any intervention from the boot switcher by using the boot-from-flash switch (switch S1-1). However, because Angel relocates itself to SDRAM there is no restriction on its location in flash. You can program it into another block if necessary.

The standalone variants of the µHAL demo program are built to run from block 64 of application flash (`0x24800000`). This can be changed by changing the read-only base address when linking the image. If the read-only base is an address in RAM the boot switcher copies the image into RAM before transferring control to it.

4.5.2 Boot switcher

The boot switcher is run if the switch S1-1 is ON. If this switch is OFF then the hardware boots directly from the first location in flash.

The boot switcher routine is embedded in the boot monitor and is the first thing that is run. It reads switch S1-4 and if it is ON passes control to the boot monitor. If it is OFF the boot switcher attempts to find and run an image in flash. This is summarized in Table 4-6.

Table 4-6 Boot switch settings

S1-1	S1-4	Action
OFF	OFF	System restarts at the first address in flash
OFF	ON	System restarts at the first address in flash
ON	OFF	boot switcher runs and searches for the boot image in flash
ON	ON	The boot monitor command interpreter runs.

4.5.3 Integrator clocks

The boot monitor stores settings for the clocks in the SIB. You can modify and display them using the boot monitor command interpreter (see *CC: Set clocks from SIB* on page 4-17).

4.5.4 LEDs

If the boot switcher is unable to find or run an image in flash, the red LED on the motherboard is illuminated.

If an attempt is made to run a µHAL image that is not compatible with the hardware (for example, an image built for an ARM720T is run on an ARM920T) the red LED flashes at a one second interval.

4.5.5 Multiple core modules

The Integrator can be fitted with up to four core modules. Each one is equipped with a processor. If more than one core module is fitted to the Integrator motherboard, the boot monitor only runs on the primary processor (on core module 0). The boot switcher runs on all processors, and all processors run the same image from flash. This image must be multiprocessor aware.

4.5.6 Loading images using the boot monitor

To load images using Motorola 32 S-record loader, you need a terminal emulator that can send raw ASCII data files. In the ARM Firmware Suite, Motorola 32 S-record images are built with the `.M32` file extension. There are prebuilt Motorola 32 S-record Angel images.

Motorola 32 S-record files can be built for other images such as, for example, the standalone µHAL demo programs, using the `FromELF` utility. Use the `-nodebug` and `-nozeropad` with `FromELF` as this produces a significantly smaller file and reduces the time required to load it.

Use the Motorola 32 S-record loader as follows:

1. Set your terminal emulator to enable XON/XOFF flow control.
2. Reset the Integrator system with both switches S1-1 and S1-4 in the ON position. This causes the boot monitor command interpreter to run.
3. At the command prompt type `L` to start the Motorola 32 S-record loader. The following dialog is displayed:

   ```
   boot Monitor > l
   Load Motorola S Records into flash
   Deleting Image 0
   Type Ctrl/C to exit loader.
   ```

   Any image the boot monitor loads is numbered image zero. If an image zero already exists, it is deleted first.
4. Use the **send file** option on your terminal emulator to download the Motorola 32 S-record image.

   The boot monitor transmits a dot for every 64 records received from the terminal emulator.
5. When the terminal emulator has finished sending the file, type `Ctrl+C` to exit the loader. On exit the loader displays the number of records loaded, the time the load took. It also lists any blocks it has overwritten.
6. Now move switch S1-4 to the OFF position and reset the system to run the image.

After the boot monitor has loaded the image, it sets the boot image number to zero. When the system restarts, the boot switcher finds and boots the last image loaded.

4.5.7 Build variants

Currently there are several different build variants for Integrator, as shown in Table 4-7.

Table 4-7 Core module images

Core module	Image type
Integrator	Runs on all supported processors
IntegratorT	Runs on all supported processors that can execute Thumb instructions
Integrator720T	ARM720T specific image
Integrator740T	ARM740T specific image
Integrator920T	ARM920T specific image
Integrator940T	ARM940T specific image

Trying to run an incompatible image, for example an Integrator720T image on an ARM940T, causes the red LED on the motherboard to flash.

4.6 Using boot monitor on Prospector

This section describes the power-on sequence for Prospector. The boot switcher is embedded in the boot monitor and is the first thing that is run. It reads switch U25-5 and if it is on, passes control to the default application (boot monitor). If it is off, the boot switcher attempts to find and run an image in flash.

4.6.1 The Prospector board

This section provides an overview of the ARM Prospector P-1100 development system. The ARM Prospector provides a flexible system for evaluation and development on a platform with a high degree of integration of:

- memory
- LCD screen controller
- timers
- interrupt controller
- power management
- removable storage.

By including all of these features, along with a reusable pool of software, Prospector allows rapid porting, evaluation, and development of derivative products.

The Prospector P-1100 includes a 190MHz SA-1100 StrongARM ASSP which has built-in controllers for DRAM, flash, and color LCD. It also includes 2 serial ports, IrDA port, power management, and separate Instruction and Data (Harvard) Caches.

The board has 32MBytes EDO DRAM and can be extended to a maximum of 64MBytes of flash. The power circuitry allows operation from 3V battery or external supply with a voltage of between 6V and 12V (9V/1.5A is recommended).

4.6.2 Flash on Prospector

Prospector has one physical area of flash that is logically divided into boot flash and application flash. Table 4-8 provides a summary of the flash areas.

Table 4-8 Flash usage on Prospector

Device	Size	Organization	Part	Usage
Boot flash	512K	2x256K blocks	Intel G28F640J5	Boot monitor and Angel images
Application flash	31.5M	126x256K blocks	Intel G28F640J5	Applications and data

Boot flash

The boot flash contains the boot monitor and switcher, and the Angel debug monitor image.

Caution

This area can be reprogrammed using BootFU. However, you must take greater care as incorrect programming can corrupt the image that runs when the system starts and might prevent the system from booting. If the boot image does become corrupt, the flash must be reprogrammed via JTAG.

Application flash

The application flash is a general-purpose area that can be used to store any images or data that require to be held in nonvolatile memory. The ARM Flash Library implements a simple mechanism for storing multiple images in flash. This structure enables the boot switcher to select and run the correct boot image. The ARM Flash Utility uses the ARM Flash Library to program and delete images in application flash. In Table B-1, a block is defined as the smallest area of flash that can be independently deleted. The flash library supports storing an image in either a single block or contiguous multiple blocks.

Location of images in flash

The normal location for Angel is block 1 in the boot flash. However, because Angel relocates itself to SDRAM, there is no restriction on its location in flash. It can be programmed into another block if necessary.

The standalone variants of the µHAL demo program are built to run from block 0 of application flash (`0x04080000`). You can change this by changing the read-only base address when linking the image. If the read-only base is an address in RAM the boot switcher copies the image into RAM before transferring control to it.

4.6.3 Start-up sequence

The boot switcher allows you to program multiple executable images into flash and provides a simple mechanism to run them. When power is applied to Prospector, the following steps occur:

1. The boot switcher code is executed. This code looks at the switch to determine whether the default application (boot monitor) or a user-selected image should be run.
2. If it is the user-selected image, the boot switcher looks for a SIB which contains the image number. Then flash is scanned for a matching image number and the image checksum is calculated and validated.
3. If the image footer indicates that it should run from RAM, then memory is initialized before the image is copied into place.
4. Control is passed to the selected image.

If the image cannot be found, or the checksum fails, control is passed back to the boot monitor, which sends an appropriate message out of the serial port before printing the banner.

4.6.4 Prospector system-specific boot monitor

The Prospector boot monitor is programmed into the boot flash as image 4280910 (`0x41524E` or `'ARM'+1`). This allows the boot switcher code to copy the image to RAM before executing it.

The boot monitor has to run from RAM in order to program data into flash, as the flash does not allow read access when programming. The top 32KB of RAM is reserved for the MMU Lookup Tables.

The Prospector provides a set of system-specific boot monitor commands. See *Prospector-specific commands for boot monitor* on page 4-22.

4.6.5 LEDs

If the boot switcher is unable to find or run an image in flash, the red LED on the motherboard is illuminated.

4.6.6 Loading images

Use a terminal emulator that is able to send raw ASCII data files to load Motorola 32 S-record images. In the ARM Firmware Suite, Motorola 32 S-record images are built with the `.M32` file extension. Motorola 32 S-record files can be built for images such as, for example, the standalone µHAL demo programs, using the FromELF utility. Use the `-nodebug` and `-nozeropad` with FromELF as this produces a significantly smaller file and reduces the time required to load it.

Use the Motorola 32 S-record loader as follows:

1. Set your terminal emulator to enable XON/XOFF flow control.
2. Reset the Prospector system with switch U25-5 in the ON position. This causes the boot monitor command interpreter to run.
3. At the command prompt type L to start the Motorola 32 S-record loader. The following text is displayed:

   ```
   boot Monitor > l
   Load Motorola S Records into flash
   Deleting Image 0
   Type Ctrl/C to exit loader.
   ```

   Any image the boot monitor loads is numbered image 0. If an image 0 already exists it is deleted first. See *L: Load S-records into flash* on page 4-7 for more information on the load command.
4. Use the **send text file** option to download the Motorola 32 S-record image.

   The boot monitor will transmit a dot for every 64 records received from the terminal emulator.
5. When the terminal emulator has finished sending the file, type `Ctrl+C` to exit the loader. On exit the loader displays the number of records loaded and the time the load took. It also lists any blocks it has overwritten.
6. Move switch U25-5 to the OFF position and reset the system to run the image.
7. After the boot monitor has loaded the image it sets the boot image number to zero. When the system restarts, the boot switcher finds and boots the last image loaded.

4.7 Rebuilding the boot monitor

Use the project files, or makefile, in the `bootMonitor` subdirectory of the source directory for your board to rebuild the boot monitor library.

For example, if you copied `windows\` contents to `C:\AFS` use `C:\AFS\source\Integrator940T\bootMonitor\Build\makefile` to rebuild the library for the Integrator board with an ARM940T processor.

For general information on makefiles and directory structure, see *AFS source structure* on page 11-4. For more detailed information on building AFS components see Chapter 11 *Building AFS Components*.

Chapter 5
Operating Systems and µHAL

This chapter describes porting an operating system to an ARM-based evaluation board that has previously had µHAL ported to it. It contains the following sections:

5.1 About porting operating systems

µHAL provides a basic API that enables simple applications to run on a variety of ARM-based development systems. You can also use it as the basis of a port of an operating system.

There are two types of operating system that can use µHAL:

- Simple threaded operating systems that run directly out of physical memory (or out of virtual memory mapped directly to physical memory). You can often link simple operating systems directly to µHAL. The operating system then functions as a µHAL application. The example of this type of operating system used in this chapter is µC/OS-II. It runs without further porting effort on any ARM evaluation board that has had µHAL ported to it. This why µC/OS-II is often the first operating system to run on a new ARM-based platform.
- Complex operating systems that utilize virtual memory (possibly using demand paging mechanisms). You cannot link these more complex operating systems directly with µHAL but they can reuse parts of µHAL. Reusing µHAL makes porting simpler than it otherwise might be. An example of this type of operating system is Linux. *Simple operating systems* on page 5-3 and *Complex operating system* on page 5-12 discuss these two types of operating system.

5.2 Simple operating systems

This section describes how ARM-specific porting code is used to initialize µC/OS-II and to allow µC/OS-II to carry out context switching.

For a simple operating system to run directly over µHAL, the following conditions must be met:

- The OS must have a fixed memory map of either physical memory or a virtual constant memory map. The memory map must be consistent with the default map defined by µHAL.
- The OS must use context switching of tasks or threads at the end of an interrupt (usually a periodic timer) or when it exits from a system call.
- The OS must be capable of being built using the ARM software development tools.

This type of operating system is isolated from the specific hardware details of the development platform because it utilizes µHAL code for system initialization, timer, and interrupt handling. It is the ARM-specific porting code that bridges the gap between the operating system and µHAL.

5.2.1 About µC/OS-II

µC/OS-II is a portable, ROM-able, preemptive, real-time, multitasking kernel that can manage up to 63 tasks. µC/OS-II is comparable in performance to many commercially available kernels. The ARM Firmware Suite includes a port of µC/OS made to the ARM architecture using the uHAL interfaces. µC/OS-II provides the following features:

- creating and managing up to 63 tasks
- creating and managing semaphores
- delaying tasks for a specified number of ticks or amount of time
- locking and unlocking the scheduler
- servicing interrupts
- changing the priority of tasks
- deleting tasks
- suspending and resuming other tasks from within a task
- managing message mailboxes and queues for intertask communications
- managing fixed-sized memory blocks
- managing a 32-bit system clock.

Note

If you wish to use µC/OS-II within a product or wish to redistribute µC/OS-II you must seek a license arrangement with Micrium Inc., the owners of µC/OS-II.

You do not have to understand µC/OS-II completely in order to understand the principles involved. If you want more information on the OS, read *Micro-C/OS-II, The Real-Time Kernel*.

5.2.2 Initializing the operating system

The entry point to a simple operating system (as it is for all other µHAL applications) is the `main()` routine. Example 5-1 shows this using the `ping.c` example program in µC/OS-II.

Example 5-1 Operating system initialization - main()

```
/*
 * Main function.
 */
int
 main(int argc, char **argv)
{
 char Id1 = '1';
 char Id2 = '2';

   OSInit(); /* needed by uC/OS */

   OSTimeSet(0);
   /*     create the semaphores    */
   Sem1 = OSSemCreate(1);
   Sem2 = OSSemCreate(1);

   /*  create the tasks in uC/OS and assign decreasing priority to them  */
   OSTaskCreate(Task1, (void *)&Id1, (void *)&Stack1[STACKSIZE - 1], 1);
   OSTaskCreate(Task2, (void *)&Id2, (void *)&Stack2[STACKSIZE - 1], 2);

      OSStart(); /* start the game */

   /* never reached */
}                              /* main */
```


When the `main()` routine is called, µHAL has already initialized the system. For example, address mapping is turned on. The operating system requires only to initialize itself and start running:

1. The first call from `main()` is to `OSInit()` that, in the case of µC/OS-II, initializes the operating system state (for example its priority map).

   Rather than clutter `OSInit()` with µHAL-specific code, `OSInit()` calls the µHAL-specific routine `ARMTargetInit()` (in `uhal.c`) to set things up. See the listing in Example 5-2 on page 5-6 for details of how `ARMTargetInit()` performs the following actions:

   a. Prints a series of messages using `uHALr_printf()`.

   b. Resets the MMU to a clean state using a call to `uHALr_ResetMMU()`.

   c. Initializes interrupt handling and timers.

   d. Defines the pre- and post-interrupt handling routines (`IrqStart()` and `IrqFinish()` respectively), that are used by µC/OS-II to context switch at the end of an interrupt.

2. `main()` creates several threads. Each thread has an area of stack that is initialized with an initial register set. The initial PC for a task contains the address of the thread routine (in this case `Task1()` and `Task2()` respectively).

3. Finally, `main()` starts the operating system with a call to `OSStart()`. As with `OSInit()`, `OSStart()` contains no µHAL-specific code but calls `ARMTargetStart()` to start the operating system, see Example 5-3 on page 5-7.

   `ARMTargetStart()` starts the system timer. The system timer functions as a periodic timer and issues an interrupt request every millisecond. When the interrupts occur, the operating system controls whether or not it requires a context switch.

   `OSStart()` selects the highest priority task that is runnable (in this case Task1) and runs it by loading its registers from its stack.

Example 5-2 Operating system initialization - ARMTargetInit()

```
#define BUILD_DATE "Date: " __DATE__ "\n"

/*  Initialize an ARM Target board */
void
 ARMTargetInit(void)
{
    /* ---- Tell the world who we are ----------------- */
    uHALr_printf("uCOS-II Running on a") ;
#if defined(EBSA285)
    uHALr_printf("n EBSA-285 (21285 evaluation board)\n") ;
#elif defined(BRUTUS)
    uHALr_printf(" Brutus (SA-1100 verification platform)\n") ;
#elif defined(INTEGRATOR)
    uHALr_printf("n Integrator board\n") ;
#elif defined(PROSPECTOR)
    uHALr_printf(" Prospector board\n") ;
#else
    uHALr_printf("n unknown ARM board\n") ;
#endif
    uHALr_printf(uHAL_VERSION_STRING);
    uHALr_printf("\n") ;
    uHALr_printf(BUILD_DATE);
    uHALr_printf("\n") ;
#ifdef DEBUG
    uHALr_printf("Initialising target\n");
#endif
    uHALr_ResetMMU(); /* ---- disable the MMU -- */
    ARMDisableInt();  /* ---- disable interrupts (IRQs------ */
    /* ---- soft vectors ------------------------ */
#ifdef DEBUG
    uHALr_printf("Setting up soft vectors\n");
#endif
    /* Define pre & post-process routines for Interrupt */
    uHALir_DefineIRQ(IrqStart, IrqFinish, (PrVoid) 0);
    uHALr_InitInterrupts();
#ifdef DEBUG
    uHALr_printf("Timer init\n");
#endif
    uHALr_InitTimers();
#ifdef DEBUG
    uHALr_printf("targetInit() complete\n");
#endif
}                               /* targetInit */
```

Example 5-3 Operating system start up

```
/* start the ARM target running */
void
 ARMTargetStart(void)
{
#ifdef DEBUG
    uHALr_printf("Starting target\n") ;
#endif

    /* request the system timer */
    if (uHALr_RequestSystemTimer(
                          PrHandler) OSTimeTick,
                         (const unsigned char *)"uCOS-II") <= 0)
        uHALr_printf("Timer/IRQ busy\n");

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

5.2.3 Context Switching

µC/OS-II switches context and causes another thread to run under the following conditions:

- when a thread makes a system call that causes it to stop running
- if an interrupt is received.

A thread might be caused to stop running when it waits on a semaphore or a timer.

A sequence of context switches for `Task1()` (listed in Example 5-8 on page 5-10) and `Task2()` (listed in Example 5-9 on page 5-10) is:

1. The call to `OSSemPend()` does not cause a context switch, and so `Task1()` prints `1+` before calling `OSTimeDly()`.

2. `OSTimeDly()` causes the `Task1()` thread to be suspended and µC/OS-II starts to run `Task2()`.

   This form of context switch involves saving the context of the current thread (all of its registers and the CPSR) on its stack and restoring the context of the highest priority task, in this case `Task2()`.

3. `Task2()` does not wait on the first call to `OSSemPend()` either. It goes on to print `[` and then calls `OSTimeDly()` that suspends `Task2()` pending the timer expiring.

4. At this point, Task1 still cannot run as its (shorter) timer has not yet expired. The output is shown in Example 5-4. (Output is done over the serial port if it is a standalone image or using the debug console if it is a semihosted image.)

Example 5-4 Initial output

```
uCOS-II Running on an Integrator board
uHAL v1.1:
Date: Aug 12 1999

1+[
```

5. Each time an interrupt occurs, the µHAL interrupt handling code `uHALir_TrapIRQ()` saves the current register set on the stack and checks for a start-of-interrupts handling routine. For µC/OS-II, this is `IrqStart()` as shown in Example 5-5.

Example 5-5 IrqStart()

```
extern int OSIntNesting;
/* This is what uCOS does at the start of an IRQ */
void IrqStart(void)
{
    /* increment nesting counter */
    OSIntNesting++;
}
```

6. `IrqStart()` increments the global count `OSIntNesting` that is used in the µC/OS-II scheduler. The µHAL interrupt handling code dispatches the timer interrupt handling code and, eventually, the µC/OS-II timer routine `OSTimeTick()` is called.

   `OSTimeTick()` decrements the delay timer of any delayed thread. This might make a task runnable. Task1 becomes runnable as soon as its delay timer expires. At the end of the µHAL interrupt handler, µC/OS-II checks for an end of interrupts handling routine. For µC/OS-II, the end of interrupt handler is `IrqFinish()` as shown in Example 5-6 on page 5-9.

Example 5-6 IrqFinish()

```
/* This is what uCOS does at the end of an IRQ */
extern int OSIntExit(void);
extern void IRQContext(void); /* post DispatchIRQ processing */
PrVoid IrqFinish(void)
{
    /* call exit routine -
       return TRUE if a context switch is needed */
    if (OSIntExit() == TRUE)
        return (IRQContext);
    return ((PrVoid) 0);
}
```

7. `IrqFinish()` calls `OSIntExit()` to determine if a context switch is necessary. If a context switch is necessary, `IrqFinish()` returns the address of `IRQContext()` (the µC/OS-II interrupt-specific context switching routine).

   Normally, the µHAL interrupt handling routine restores the saved registers from the stack and returns from the interrupt. Because the end-of-interrupt routine returned an address, `IrqFinish()` calls the µC/OS-II interrupt context switching routine with the saved registers still on the stack.

   Note

   The usage of registers on the stack must be the same for both µC/OS-II and µHAL.

8. When `Task1()` runs it prints `1-`, posts to the `Task2()` semaphore (incrementing it) and waits on its own semaphore.

9. When `Task2()` is selected to run (after its delay timer expires), it prints `2]` and posts to the `Task1()` semaphore. This allows `Task1()` to run, causing the whole cycle to repeat as shown in Example 5-7:

Example 5-7 Later output

```
uHAL v1.1:
Date: Aug 12 1999

1+[1-2]1+[1-2]1+[1-2]1+[1-2]1+[1-2]1+[1-2]1+[1-
```

Example 5-8 shows the code for Task1.

Example 5-8 Context switching Task1()

```
void
 Task1(void *i)
{
   uint Reply;
   for (;;)
   {
      OSSemPend(Sem2, 0, &Reply);  /* wait for the semaphore  */
      uHALr_printf("1+");
      OSTimeDly(100);              /* wait a short while */
      uHALr_printf("1-");
      OSSemPost(Sem1);             /* signal the semaphore */
   }
}
```

Example 5-9 shows the code for Task2.

Example 5-9 Context switching Task2()

```
void
 Task2(void *i)
{
    uint Reply;
    for (;;)
    {
      OSSemPend(Sem1, 0, &Reply); /* wait for the semaphore */
      uHALr_printf("[");
      OSTimeDly(1000);            /* wait a short while */
      uHALr_printf("2]");
      OSSemPost(Sem2);            /* signal the semaphore */
    }
}
```

5.2.4 Efficiency considerations

The method of switching context during an interrupt is not particularly efficient because it involves several calls into C code. It does, however, have the advantage of being highly portable.

If efficiency is a constraint, the port of an operating system can use its own interrupt handling code instead of the µHAL routines:

- An initial step to improving efficiency is to replace the µHAL interrupt handler `uHALir_TrapIRQ()` but still call the C-based interrupt and timer handling routines provided by µHAL. If the replacement interrupt handler uses the `READ_INT` macro, it is not dependent on the version of the ARM evaluation board that is used. This also has the advantage that the stack usage can differ between the operating system and µHAL.
- You can make additional improvement if, once a timer is started, it is periodic and does not require any further intervention. The operating system can reuse the interrupt and no additional calls to µHAL are required once the timer is running.
- The final option is to reuse parts of µHAL in a board-specific port and tailor the code to the operating system. This approach is not very portable, but you can use it to improve efficiency.

5.3 Complex operating system

Complex operating systems cannot directly use µHAL but they can do one or more of the following:

- reuse its definitions and some of its board-specific code
- use a µHAL-based image as a loader or initializer.

5.3.1 Reusing definitions

Reusing definitions usually means using the definitions from `platform.h`. This gives the operating system definitions of where in the physical memory map registers are located, as well as bit settings for those registers. For example, `platform.h` for the Integrator platform defines the physical address of the debug register set as:

```
#define INTEGRATOR_DBG_BASE       0x1A000000
```

This definition can be directly used if the address map remains physical.

If the operating system runs out of virtual memory, there must be a further definition. Using Linux as the example, the port of Linux to the Integrator platform maps all of the Integrator registers to virtual address `0xF0000000` and places them closer together using the following definition:

```
#define IO_BASE 0xF0000000
#define IO_ADDRESS(x)  ( (x>>4) + IO_ BASE )
```

This means that the virtual address of Integrator debug registers becomes `((0x1A000000 >> 4) + 0xF0000000)` or `0xF1A00000`. In the port of Linux to the Integrator platform, each bank of registers is mapped to its own virtual address. The LED offset of from the debug base is defined in `platform.h` as `INTEGRATOR_DBG_LEDS_OFFSET` and has a value of 4 bytes. This means that the LEDs can be found using this address:

```
IO_ADDRESS(INTEGRATOR_DBG_BASE) + INTEGRATOR_DBG_LEDS_OFFSET
```

The bit settings can then be used as normal. For example, bit 0 is the green LED. There are also defines for the LEDs, `GREEN_LED` for example.

5.3.2 µHAL-based loader application

A pure µHAL application is used to initialize the system and then to load the operating system. The binary image of the Linux kernel, for example, is loaded into flash using the ARM Flash Utility.

A µHAL-based loader application initializes the Integrator PCI subsystem and then copies the kernel into memory before transferring control to it. The Linux kernel does not itself contain any PCI setup code, instead it scans the PCI subsystem discovering how the µHAL application set it up. This removes the need for the Linux kernel to understand the details of setting up the V3 PCI chip and how to route interrupts on the Integrator platform.

You can also modify the µHAL loader/initialization application to pass a data structure to the operating system kernel that describes the system.

Chapter 6
Angel

This chapter describes the function of Angel on development boards, and how µHAL and Angel debug monitor sources are related. The code required to port Angel to a system that already has µHAL is covered in detail. The chapter contains the following sections:

For more information on using Angel with a debugger, see the documentation provided with SDT 2.5 or ADS 1.0.

6.1 About Angel

This section describes how µHAL and the Angel debug monitor sources are related. It recommends a number of coding practices that allow Angel and µHAL to be quickly and easily ported to ARM-based systems, and for semihosted µHAL applications to run with Angel.

The Angel debug monitor uses a serial line or Ethernet to communicate with a development host running an ARM debugger. The debugger uses the *Angel Debug Protocol* (ADP) to send requests to Angel to, for example:

- download images
- set breakpoints
- examine registers and variables.

These functions are described in detail in the documentation supplied with your debugger. To carry out these functions, Angel uses the physical system resources, such as interrupts, serial ports, and memory (for stack and context storage). When Angel changes between ARM and Thumb state, it saves and restores context.

Building a fully functional µHAL-based Angel is simplified if you take a series of small steps. This is where µHAL is used. Port µHAL to your platform first and verify that:

- memory management is functioning correctly
- LEDs are functioning
- the serial port is operating
- interrupts are being generated.

These steps can be performed one at a time. When you have verified that the board is functioning correctly at this level, re-use the code within Angel.

It is usually better to build on an example of an existing port, than to start again. There are currently two ports of Angel in ARM Firmware Suite v1.0 that are based on reusing µHAL sources. These are targeted at the Integrator and the Prospector development systems. This chapter uses both of these sources as examples.

6.1.1 Angel and cache memory

On some development boards, Angel does not work when cache memory is enabled. If this is the case for your development board, disable the cache when running Angel. Your applications will run several times slower than with cache. However, the debugging process will not otherwise be affected.

If you want your applications to run at full speed with cache memory, use Multi-ICE instead of Angel.

6.1.2 Angel and RAM memory

Depending on the development board you are using, you may have to add more memory if you are using Angel to debug applications.

Integrator boards

The prebuilt Angel as supplied on the ARM Firmware Suite CD runs on the internal SRAM on the Integrator/AP and /SP.

Prospector boards

The standard RAM is sufficient to run Angel. The RAM on Prospector cannot normally be upgraded.

6.1.3 Thumb support

The prebuilt Angel image and the default Angel build support Thumb programs. Switch Thumb support on using the `THUMB_SUPPORT=1` define.

6.1.4 Using Angel with a debugger

Once you have installed Angel into the flash memory, you can use it with your debugger. The way you connect to Angel depends on the debugger you are using:

ADW/ADU for SDT 2.50

See the SDT 2.50 *User Guide* for ADW/ADU.

armsd The command line must be of the form:

```
armsd -adp -port s=1 -linespeed 38400 image.axf
```

ADW/AXD for ADS 1.0

See the *Debuggers Guide* supplied with ADS.

You can test whether Angel has installed by setting a terminal emulator to 9600 baud, setting the Integrator switch to boot the Angel image, and resetting the board. Angel attempts to communicate with the debugger over the serial port. The terminal emulator displays some symbols and then the Angel banner.

6.1.5 Downloading Angel to a development board

Some development boards come with Angel installed. If your board does not have Angel already installed, you must download the Angel image for your board and processor.

6.2 Angel on Integrator

This section provides an overview of Angel on the ARM Integrator development system.

6.2.1 Location in memory

The actual address it is linked at is `0x028000000` which is the motherboard SSRAM.

6.2.2 Caches

Angel for Integrator runs without enabling caches. To get the maximum performance from the system, you must enable caches. This requirement on the application is also true for Multi-ICE. See *uHALr_EnableCache()* on page 3-12 for details on enabling caches.

Applications built against µHAL can use µHAL functions to control the cache. See *Simple API MMU and cache functions* on page 3-11 and *Extended API MMU and cache functions* on page 3-40.

6.2.3 Line speed

The maximum line speed that Angel will support depends on factors such as the clock settings for the processor and buses. A maximum line speed of 57,600Kbps is supported for a system with the optimum clock settings and caches disabled (Angel does not support cached code).

6.2.4 Downloading Angel

Use the boot monitor and Angel to load and debug programs over a serial port.

Preparing the board

Follow these steps to prepare you 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 with the settings as shown in Table 6-1.

Table 6-1 Serial port settings

Property	Value
Bits per second	38400
Data bits	8
Parity	None
Stop bits	1
Flow control	Xon/Xoff

   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. Set switches S1-1 and S1-4 to the ON position and S1-2 and S1-3 to the OFF position.
   e. 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 Example 6-1:

Example 6-1

```
ARM bootPROM [Version 1.0] Rebuilt on Aug  6 1999 at 14:18:53
Running on a Integrator (Board revision v1.0, ARM740T 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 >
```

   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.

3. Use the boot monitor to download one of the prebuilt Angel images from the CD to the development board. See Chapter 4 *ARM Boot Monitor* for details on the boot monitor commands and an example of downloading an image.
   a. At the command prompt type `L` to start the Motorola 32 S-record loader.
   b. Use the terminal emulator to download the image. The Angel image for the Integrator is `angIntegrator.m32` and is located in:

```
\common\images\Integrator\angel\Integrator.b\gccsunos\little_rel\
```

   c. 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.
   d. On exit, the loader will display the number of records loaded, the time the load took, and any blocks it has overwritten.

4. Configure your debugger to use the Angel debug agent.

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

   ADW/ADU for SDT 2.50

   See the *SDT User Guide*.

   armsd The command line should be of the form:

```
armsd -adp -port s=1 -linespeed 38400 image.axf
```

   ADW/AXD for ADS 1.0

   See the *Debuggers Guide* supplied with ADS.

6.3 Angel on Prospector

This section provides an overview of Angel on the ARM Prospector P-1100 development system.

6.3.1 Location in memory

Angel is linked to run from DRAM. The actual address it is linked at is `0x01FE8000` which is near the top of the 32MB DRAM region. The top 32KB of DRAM is reserved for the MMU Lookup Tables and the Angel executable occupies the 64KB below that.

6.3.2 Caches

Angel for Prospector runs with caches and MMU enabled. This allows applications to get the maximum performance from the system. Use the µHAL functions to control the cache, see *Simple API MMU and cache functions* on page 3-11 and *Extended API MMU and cache functions* on page 3-40.

6.3.3 Line speed

The maximum line speed that Angel will support depends on factors such as the clock settings for the processor, and on whether caches are enabled. Running on a 190MHz system with caches enabled, a maximum line speed of 115,200Kbps is supported.

6.3.4 Initial loading of Angel into flash

Angel is supplied in the boot area of flash as image 911. Because Angel has the facility to relocate itself to DRAM, the BootFU and AFU utilities can assign any image number to it and relocate it in flash.

6.4 µHAL-based Angel

Most of the Angel code is the same for µHAL-based Angels as any other (non-µHAL) Angel. The only difference is that µHAL-based Angels utilize system-dependent code that is held within µHAL. Angel still requires the same set of macros, source code, and definitions that it did before, but now it imports some of these definitions from µHAL source files.

You can download one of the prebuilt Angel images or rebuild Angel and download your modified version.

If you rebuild Angel, the version string is `Unreleased`. This indicates that the copy of Angel was built outside of ARM. You can edit the version string information in the source files in `banner.c` and `banner.h` to display your own version code.

6.4.1 Source directory for Angel

A particular board is supported by code held in the relevant board-specific directory of µHAL. For example, the sources that relate to Integrator are all held in the directory `uHAL\Boards\INTEGRATOR`. This set of sources consists of:

`platform.h` **and** `platform.s`
: These files contain definitions of the board, including its memory layout, and devices.

`driver.s`
: This file contains low-level assembly code needed for the board to function.

`target.s`
: This file contains ARM assembly macros that are used within µHAL. For example, to switch the memory map or light a particular LED. Routines in `driver.s` often use these macros too.

`memmap.s`
: This file describes (in tabular form) the memory map for a particular system. This includes where in virtual memory, the various areas of physical memory are mapped.

`board.c`
: This file contains C routines that support the operation of the board. For example, the PCI Configuration space access routines for Integrator are stored here.

6.4.2 Angel sources and definitions

A µHAL-based Angel requires extra sources and definitions. For example, the `angel\Integrator` directory contains:

`banner.h` This file provides the startup banner displayed by this Angel.

`devices.c` This file describes the set of devices available to Angel for this board.

`makelo.c` This file allows variables to be shared between both the `.c` and the `.s` assembler files. It produces an assembler header file called `lolevel.s`.

`timerdev.c`

This file implements Angel timers for the Integrator platform. This timer is used for profiling and for polled device drivers (for example, Ethernet devices).

`integrator.h`

This file contains Integrator-specific Angel definitions, including `platform.h`.

`devconf.h` This file is the main configuration file of the target image. It describes the uses that Angel makes of the system resources (for example stack memory).

`serial.c` This file implements the serial driver for Angel on this system.

`ambauart.h` This file contains definitions for the serial interface hardware.

The `angel\Propector` contains similar files. The use of these files by Angel is described in the Angel documentation supplied with SDT 2.5 or ADS 1.0.

6.5 Building a µHAL-based Angel

If you are building a µHAL-based Angel, not all of the code is in the Angel board directory (such as `angel/Integrator`). Part of the code is located in the board-specific or processor-specific area of µHAL (for example, `AFS/source/all/uHAL/Boards/INTEGRATOR` or `AFS/source/all/uHAL/Processors/ARM720T`). This means that your project file or makefile must point to these directories in order to obtain those sources and include files. See *Building the µHAL library* on page 2-8 and Chapter 11 *Building AFS Components* for an overview of the build process.

The following examples are from the makefile for the Integrator Angel. Example 6-2 defines where the various parts of code that Angel is dependent on are located.

Example 6-2 Defining code locations

```
ADS_BUILD=0
TARGET  = Integrator
IMAGE  = angIntegrator
ROOTDIR = ../..
UHAL_BASE = $(ROOTDIR)/../uHAL/
UHAL_BOARD_DIR = $(UHAL_BASE)/Boards/INTEGRATOR
TARGDIR = $(ROOTDIR)/Integrator
ETHDIR  = $(ROOTDIR)/ethernet
PROCESSOR = ARM7T
```

The part of the makefile shown in Example 6-3 on page 6-11 sets up the flags to be used with the assembler (a similar definition is needed for the compiler). It ensures that the appropriate µHAL directories are searched for include files.

Example 6-3 Setting the assembler flags

```
AFLAGS= -g -apcs $(APCS) $(ASENDIAN) -arch 4 \
    -I$(OBJDIR) \
    -I$(ROOTDIR) -I$(TARGDIR) -I$(UHAL_BOARD_DIR) -I$(CLIB)\
    -I$(UHAL_BASE)/h \
    -I$(UHAL_BASE)/Processors/$(PROCESSOR) \
    -PD "LOGTERM_DEBUGGING SETA $(LOGTERM_DEBUGGING)" \
    -PD "ANGELVSN SETA $(ANGELVSN)" \
    -PD "DEBUG SETA $(DEBUG)" \
    -PD "LATE_STARTUP SETA $(LATE_STARTUP)" \
    -PD "ROADDR SETA $(ROADDR)" \
    -PD "THUMB_SUPPORT SETA $(THUMB_SUPPORT)" \
    -PD "ASSERT_ENABLED SETA $(ASSERT_ENABLED)" \
    -PD "MINIMAL_ANGEL SETA $(MINIMAL_ANGEL)" \
    -PD "ETHERNET_SUPPORTED SETA $(ETHERNET_SUPPORTED)" \
    -PD "DEBUG_BASE SETA $(TASKLOG_BASE)" \
    -PD "DEBUG_SIZE SETA $(TASKLOG_SIZE)" \
    -PD "$(PROCESSOR) SETL {TRUE}" \
    -PD "ADS_BUILD SETA $(ADS_BUILD)"
```

6.5.1 Angel project and makefiles

There are SDT and ADS project files and Unix makefiles in the Angel build directories.

PC project files

You can build Angel with SDT 2.5 project manager files (`.apj`) or ADS 1.0 CodeWarrior project files (`.mcp`).

Unix makefile

The CD has a makefile for use on a Unix workstation (`unix/source/all/angel/makefile`) that rebuilds versions of Angel for all target boards.

There are also makefiles that rebuild Angel for a single development board. The makefile for Integrator is `unix/source/Integrator/angel/makefile`.

If you copy the files from the CD directory onto your workstation, you must maintain the hierarchy of the CD directories. The makefile defines ROOT as the root of the build tree and is needed by `mk`. TOOLS is the tools directory that contains build tools of various kinds. The definitions in makefile are:

```
ROOT=..
TOOLS=../tools
MK= $(TOOLS)/mk
```

For general information on makefiles and directory structure, see *AFS source structure* on page 11-4.

Output formats

The Angel build creates both BIN and M32 format images. For Integrator, you can find these images in the `Integrator.b\gccsunos\little_rel` subdirectory of the `angel` directory. The names of these files are `angIntegrator.bin` and `angIntegrator.m32` respectively.

6.6 Source file descriptions

This section describes the source files used by the µHAL Angel and provides examples of their use.

6.6.1 banner.h

This file displays the banner when Angel boots. The display is output to serial port or the console window of the ARM debugger. It is good practice to use the banner to convey useful information about the system. In Example 6-4, `banner.c` displays:

- the version of Angel
- the board it is running on
- whether or not the MMU, caches, and write buffer are enabled
- the interrupt source this Angel is built to use.

Example 6-4 Using banner.h

```
#if CACHE_SUPPORTED
# define MMU_STRING " MMU on, Caches enabled, "
#else
# define MMU_STRING "MMU on, Caches disabled, "
#endif

#if ENABLE_CLOCK_SWITCHING
# define CSW_STRING " Clock Switching on "
#else
# define CSW_STRING "Clock Switching off "
#endif

#if HANDLE_INTERRUPTS_ON_IRQ
#define INTERRUPTS_STRING "(IRQ), "
#else
#define INTERRUPTS_STRING "(FIQ), "
#endif

#define ANGEL_BANNER \
"Angel Debug Monitor for Prospector " INTERRUPTS_STRING
MMU_STRING CSW_STRING "(serial)\n" \
TOOLVER_ANGEL " rebuilt on " __DATE__ " at " __TIME__ "\n"
```

6.6.2 devices.c

This file describes the set of devices that Angel has access to on this system. Example 6-5 describes the system as having a serial device and, optionally, an ethernet and debug communications channel.

Example 6-5 Using device.c (1)

```
const struct angel_DeviceEntry *const
angel_Device[DI_NUM_DEVICES] =
{
    &angel_AMBAUARTSerial[0],
#if (AMBAUART_NUM_PORTS > 1)
    &angel_AMBAUARTSerial[1],
#elif DEBUG && LOGTERM_DEBUGGING
    &angel_NullDevice,
#endif

#if ETHERNET_SUPPORTED
    &angel_EthernetDevice,
#endif

#if DCC_SUPPORTED
    &angel_DccDevice,
#endif
};
```

Example 6-6 describes the set of interrupt handlers that this Angel uses. `Angel_TimerIntHandler` is a timer interrupt for example.

Example 6-6 Using device.c (2)

```
/*
* The interrupt handler table - one entry per handler.
* DE_NUM_INT_HANDLERS must be set in devconf.h to the number of
* entries in this table.
*/

#if (DE_NUM_INT_HANDLERS > 0)
const struct angel_IntHandlerEntry
angel_IntHandler[DE_NUM_INT_HANDLERS] =
{
    { angel_AMBAUARTIntHandler, DI_AMBAUART_A }
```



```
#if (AMBAUART_NUM_PORTS > 1)
    ,{ angel_AMBAUARTIntHandler, DI_AMBAUART_B }
#elif DEBUG && LOGTERM_DEBUGGING
    ,{ angel_LogtermIntHandler, DI_AMBAUART_B }
#else
    ,{ angel_NodevIntHandler, 0}
#endif

#if TIMER_SUPPORTED
    ,{ Angel_TimerIntHandler, 0 }
#endif
};
#endif

/*
* The poll handler table - one entry per handler
*
* DE_NUM_POLL_HANDLERS must be set in devconf.h to the number
* of entries in this table.
*/
#if (POLLING_SUPPORTED && DE_NUM_POLL_HANDLERS > 0)

const struct angel_PollHandlerEntry
angel_PollHandler[DE_NUM_POLL_HANDLERS] =
{
#if DCC_SUPPORTED
    { (angel_PollHandlerFn)dcc_PollRead, DI_DCC,
    (angel_PollHandlerFn)dcc_PollWrite, DI_DCC },
#endif
};
```

6.6.3 makelo.c

This file provides a translation between C `#defines` and assembler constants. There must be a line in the `makelo.c` for every definition that the board (or Angel) code requires to be available to assembly code. The line in Example 6-7 makes the symbol `Angel_FIQStackOffset` available in assembler sources.

Example 6-7

```
fprintf(outfile, "Angel_FIQStackOffset\t\tEQU\t0x%08X\n",
         Angel_FIQStackOffset);
```

6.6.4 timerdev.c

This file makes a timer available to Angel by providing a set of plug-in routines that manage a timer. This allows Angel to:

- initialize the timer
- start the timer
- stop the timer
- get and set the timer interval.

Note

The Integrator Angel uses timer number 2.

6.6.5 serial.c

This file contains the serial device driver for this particular platform. For µHAL-based Angel, this usually reuses the board-specific definitions from µHAL (for example, the bits in the individual UART registers) to implement the serial device functions of the Angel.

6.6.6 target.s

Angel must have various macros defined within `target.s`. These are used at system startup by `startrom.s`. These macros are:

`UNMAPROM` This macro is called by the `startrom.s` ROM initialization code. It is called in systems that use ROM remapping to ensure that the ROM image is at 0 at reset. After the system has been initialized, this macro is called to switch the ROM to its physical address and RAM at 0. The actual mechanism for performing the remap varies from board to board. For more details, refer to the documentation for your hardware.

`STARTUPCODE`

This macro is called from `startrom.s` for target-specific startup. It is likely to include memory sizing, initialization of memory controllers, and interrupt controller reset.

`INITMMU` This macro initializes the MMU (or MPU).

Note

Take care with this macro because the location of the page table is important to the operation of the macro and must be given correctly. There is also a setup issue if the operation of the system is big-endian as the MMU is responsible for the byte order of the core and must be set up early to allow the correct operation of the code.

`INITTIMER` This macro allows initialization of any timers required by the application. It is called after the interrupts are disabled and the system is set in Supervisor mode.

`GETSOURCE` This macro is called by `suppasm.s` routines (the general Angel support routines). It defines the Angel interrupts used and offers a small amount of prioritization to ensure that the debug comms source has priority for Angel operation. The routine places the C-defined source value (defined in `devconf.h`). These values are used by the interrupt handler for a jump table holding the individual Angel Interrupt source handler function pointers.

`CACHE_IBR` This macro is called from `suppasm.s` support code to set an Instruction Barrier Range. This is required on systems with processors that have a Harvard cache.

None of these macros are used within µHAL and so must be written for Angel. However, existing µHAL macros can be reused. For example, the Integrator `STARTUPCODE` macro reuses the µHAL `INIT_RAM` and `DISABLE_INTS` macros.

6.6.7 devconf.h

This file is the main configuration file for the target image. It sets up the Angel resources for the specific target and shows the hardware available for Angel usage including:

- available memory map
- interrupt operation
- peripherals
- devices.

Caution

DCC and CACHE support are processor-dependent. Declaration of either of these support calls enable routines that will only work for specific processor options. If the processor options do not match your board, Angel halts.

The DEBUG_METHOD is only applicable when the DEBUG compiler option is set in the makefile. It defines the channel to be used to pass the debug messages. The definitions from the Integrator version are shown in Example 6-8:

Example 6-8

```
/* Choose the method for debug output here.  Options supported
for PID are:
*      panicblk        panic string written to RAM
*      logserial       via Serial port at 115200 baud
*      logterm         as logserial, but interactive. */
#if DEBUG
#if MINIMAL_ANGEL
#define DEBUG_METHOD panicblk
#else
#define DEBUG_METHOD logterm
#endif
#endif
```

Interrupt operation is selectable for Angel allowing the use of IRQ, FIQ or both interrupts as sources for the ADP channel communications. µHAL-based Angel currently only supports FIQ. If the FIQ is chosen as the source for Angel communications channel, the FIQ safety-level descriptor defines the operation of the FIQ with regard to use of the Angel serializer. The recommended default setting is to ensure that FIQs use the serializer and lock mechanisms. The other options are shown in `serlock.h` in the generic code section.

The memory map must be defined to allow the debugger to control illegal read/writes using the `PERMITTED` checks. These should reflect the permitted access to the system memory architecture. For Integrator, the `PERMITTED` macros are:

```
 /* These macros are used by debugger-originated memory
reads/writes to check if the write is valid. */
#define READ_PERMITTED(__addr__) (1)
#define WRITE_PERMITTED(__addr__) (1)
```

Note

You must take care with systems that have access to the full 4GB of memory, as the highest section of memory should equate to `0xFFFFFFFF` when the base and size are defined as a sum, and it may wrap around to 0. For example, if there is memory-mapped input/output at `0xFFD00000` the definition should be:

```
#define IOBase (0xFFD00000)
#define IOSize (0x002FFFFF)
#define IOTop (IOBase + IOSize)
```

and not:

```
#define IOBase (0xFFD00000)
#define IOSize (0x00300000)
#define IOTop (IOBase + IOSize)
```

By default, Angel checks for the highest available memory location from the default location. This is useful for systems, such as Integrator, where memory can be added into the DRAM slots but still must be accessed by Angel. It allows the stacks and heap more space by relocating to the top of memory. It allows a single Angel to be used across a common product range with similar memory maps but different memory sizes.

The stacks must be defined for all processor modes that are used by Angel. These always include User, SVC, UNDEF, and the appropriate mode for the chosen Interrupt source. The location of the stacks can be fixed, or can be set to the top of memory once this has been defined by the memory sizing function. All other Angel-defined memory spaces (fusion stack heaps, profile work area, application stacks) can be defined to sit relative to the stacks, or they can be given fixed locations. The default for the application heap space is above the run-time Angel code and the available space is to the lowest limit of the stacks. The definition for Integrator is shown in Example 6-9:

Example 6-9

```
/* The following are the sizes of the various Angel stacks */
#define Angel_UNDStackSize      0x0100

#define Angel_ABTStackSize      0x0100

#define Angel_AngelStackSize    (POOLSIZE *Angel_AngelStackFreeSpace)

#define Angel_SVCStackSize      0x0800
```

Note

Angel stack space is different from the application stack space to allow Angel to debug code that has corrupt or missing stacks.

The download agent area should be a spare area of RAM that can be used for testing. The download agent usually executes from the load agent address and copies itself over the resident RAM Angel image (that is, it executes in the same way as the ROM-based image).

The available devices must be defined in the structure `DeviceIdent`. The definition for Integrator is shown in Example 6-10:

Example 6-10

```
typedef enum DeviceIdent
{
    DI_AMBAUART_A,
#if (AMBAUART_NUM_PORTS > 1) || (DEBUG && LOGTERM_DEBUGGING)
    DI_AMBAUART_B,
#endif
#if ETHERNET_SUPPORTED
    DI_ETHER,
#endif
#if DCC_SUPPORTED
    DI_DCC,
#endif
    DI_NUM_DEVICES
}
DeviceIdent;
```

You must ensure that the order in this structure is the same as that defined in the array in `devices.c`, as this allows access to the register base of the specified ports in the defined order. This is also true for the interrupt handler structure. Because this is the basis for the jump table in `suppasm.s`, the order and number must be the same as defined in `devices.s`. The labels must also be placed in `makelo.c` to ensure that they are available for `suppasm.s`.

6.7 Device drivers

These files are the main area of the porting operation. The files are application-dependent. The control of the device is carried out through function pointers defined in `devclnt.h`, `devdriv.h` and `serring.h`.

The main controlling functions are:

- *angel_DeviceControlFn()*
- *Transmit control (ControlTx)* on page 6-22
- *Receive control (ControlRx)* on page 6-22
- *Transmit kickstart (KickStart)* on page 6-22
- *Interrupt handler* on page 6-23.

6.7.1 angel_DeviceControlFn()

This controls the device by passing in a set of standard control values that are defined in `devices.h` in the main directory.

Syntax

```
DevError angel_DeviceControl(DeviceId  devID, DeviceControl op,
                             void *arg)
```

where:

devID is the index of the device to control.

op is the operation to perform.

arg is a parameter depending on the operation.

Examples of the values for *devID* are:

`DC_INIT` Specific device initialization at the start of a session.

`DC_RESET` Device reinitialization to set the device into a known operational state ready to accept input from the host at the default setup.

`DC_RECEIVE_MODE`

Receive Mode Select. Sets the device into and out of receive mode.

`DC_SET_PARAMS`

Set device operational parameters. Sets the device parameters at initialization. This is also used if the host must renegotiate the parameters, for example in the instance of a change of baud rate.

Return value

Returns one of the following:

`DE_OKAY`	Control request is underway.
`DE_NO_DEV`	No such device.
`DE_BAD_OP`	Device does not support operation.

6.7.2 Transmit control (ControlTx)

When in operation, Angel defaults to the receive active state. This allows quick response to host messages. This function controls the transmit path of the serial driver, switching it on or off depending on the flag status set up in the calling routine.

6.7.3 Receive control (ControlRx)

This function is similar to *Transmit control (ControlTx)*. It controls the receive channel.

6.7.4 Transmit kickstart (KickStart)

As Angel generally operates in receive active mode, transmission must be initiated by this function. The ADP construction code sets up the bytes to be transmitted for a message to the host in a transmit buffer. It then calls the `kick_start()` function to initiate the transfer. This routine takes the first character from the transmit buffer and passes it to the serial Tx register. This causes a Tx interrupt from which the interrupt handler passes the remainder of the buffer as each character is transmitted.

6.7.5 Interrupt handler

The interrupt handlers are generic for each peripheral. In the case of the ARM development boards, the interrupt handler controls interrupts from each serial driver Tx and Rx as well as the parallel reads.

The interrupt handler determines the source of the interrupt and performs the appropriate action depending on the source:

Tx Pass bytes from the internal Tx buffer to the serial Tx FIFO, if there is space in the FIFO.

Rx Pass the byte received at the Rx FIFO into the internal Rx buffer, ready for Angel to unpack the message when the transfer is complete.

Parallel The parallel port is polled to pass the data received into the memory location requested.

All the above operations are serialized by Angel to ensure that they are not interrupted by any other operations. Interrupts are disabled from the start of the interrupt handler routine until the serializer function is called.

Other system drivers (Ethernet/DCC for example) might not require the full operation functions and instead require only a pure Rx/Tx control.

Chapter 7
Flash Library Specification

This chapter provides the complete functional specification of the ARM flash library and the various ways it can be used.

This chapter contains the following sections:

See also Chapter 4 *ARM Boot Monitor* and Chapter 8 *Using the ARM Flash Utilites* for additional information about images in flash memory.

7.1 About the flash library

Current ARM development boards (such as Integrator and Prospector) contain a large area of flash memory. This space is used to store many programs and associated data in a block structure, as defined in *Footer structure* on page 7-8.

The flash library divides the large flash memory structure into discrete blocks. In the case of the Integrator board, these are 128KB blocks. An image can contain any number of blocks, but it must conform to the flash library definition. Figure 7-1 shows the standard flash library image storage layout.

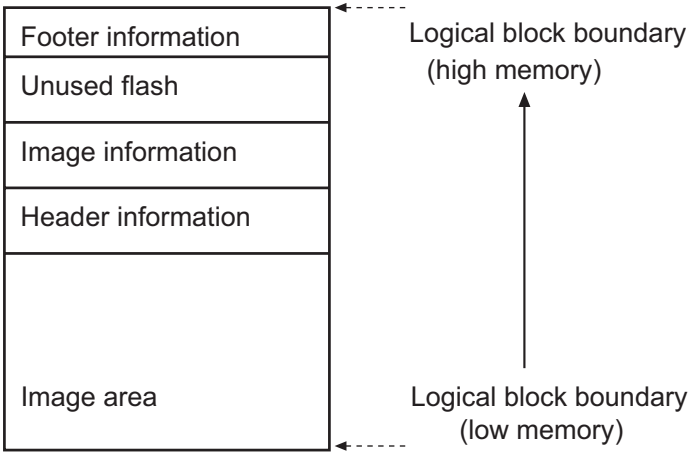

Figure 7-1 Flash library image storage layout

The following list describes the areas contained in Figure 7-1:

Image area

All of the code and read-only data segments of the image.

Header information

Any file header information from the downloaded file is placed after the image. (Not all images have header information.)

Image information

Added by the flash library code for image identification and code operations.

Unused flash

The footer must be at the end of the block of flash memory. The memory between the end of the image information and the footer is unused. If there is not room in the block containing the image for the footer, the footer will be placed at the end of the next block.

Footer information

A five-word information block containing:

- the address of the information block for this image
- the base address of the data (the start might be at the beginning of a previous block rather than at the start of this block)
- a unique 32-bit value to aid in fast searching
- the image type (that is a block an image, an SIB, or data)
- a checksum for the footer information (over the first four words only).

7.2 About flash management

Many modern embedded systems incorporate large areas of static programmable memory and they require a mechanism to allocate, program, and pass control to images of varying size. This section describes the mechanism for programming flash, how multiple images are programmed into flash, and how images are selected for automatic execution.

The main characteristics of the *ARM Flash Utility* (AFU) and ARM flash library structure are:

- The library images use a footer rather than a header. For executable images and structured data, the existence of a header complicates using the image.
- The library is managed in standard, small block sizes that hide detail from the normal user. A size of 16K provides flexibility.
- The AFU supports the following multiple-image formats:
  - ELF
  - AIF
  - binary
  - Motorola S-record.

  Binary images require additional information to be defined in addition to the data contained in the file. AFU automatically identifies the file type from the image it receives.
- The AFU application is able to read an image into memory without necessarily copying it into flash. If, for example, an invalid address is given, the image can be programmed to the correct address without having to download it again.
- The AFU removes AIF headers from binary AIF files. This ensures that the first word of the image is real image data, as it would be in a final product. This also applies to other format headers where possible.
- Images occupy sequential flash locations in order to avoid problems with image bitmasks.

Note

Scatter loading can only be implemented using the flash library if you ensure that all load regions specified in the scatter description file map to sequential blocks.

7.3 ARM flash library specifications

This section discusses the general uses for the flash library and the flash types it supports. It also describes how image management is performed in the following sections:

- *Code portability*
- *Accessing flash*
- *flashType structure* on page 7-6
- *Flash types* on page 7-7
- *Image management* on page 7-8.

The boot monitor also uses flash memory to store information about images. These *System Information Blocks* (SIBs) are application-specific. For information on the SIBs, see Chapter 4 *ARM Boot Monitor*.

7.3.1 Code portability

The flash programming library provides a standard access mechanism. The building blocks for common flash types simplify porting by:

- declaring where the flash memory is located
- identifying the type of flash
- declaring the size of the flash
- linking with the library.

The library guarantees a common access mechanism between the boot switcher, AFU, and any other programming mechanism (such as the flash downloader in the SDT).

7.3.2 Accessing flash

Primary routines are supplied that allow access to on-board flash and allow an application to:

- check that there is actually flash at a given location
- write a word
- read a word
- erase a block of flash
- program a block of flash.

Note

Caches must be disabled for the flash being programmed. Also, some systems require that the entire flash part is inactive during programming. In these cases, the flash management code (and the debug agent) must not be executing from flash.

7.3.3 flashType structure

At the lowest level, the flash memory is accessed in a uniform manner using the `flashType` device structure. Table 7-1 lists the contents of the device structure of the flash library.

Table 7-1 Device structure routines

Field	Size (in (bytes)	Value/usage
`Base`	4	Base address of this flash device.
`Size`	4	Size of flash, in bytes.
`Type`	4	Atmel, Intel, other CFI manufacturers, or unknown type.
`writeSize`	4	Size of the physical flash block when writing data. Many devices can be programmed much faster using a block-programming algorithm.
`eraseSize`	4	Size of the physical flash block when erasing data. Some devices support different erase/write block sizes
`write()`	4	Pointer to a routine to write one 32-bit word to flash.
`writeBlock()`	4	Pointer to a routine to write a block of `writeSize` bytes to flash.
`read()`	4	Pointer to a routine to read one 32-bit word from flash.
`readBlock()`	4	Pointer to a routine to read a block of `writeSize` bytes from flash.
`erase()`	4	Pointer to a routine to delete a block of `eraseSize` from flash.
`init()`	4	Pointer to a routine to lock flash to prevent erasure or programming.
`close()`	4	Pointer to a routine to unlock flash to allow erasure and programming.
`Info`	64	An ASCII string, used to identify the device (NULL-terminated).
`Next`	4	Pointer to the next flash device structure.

The library defines a C structure, shown in Example 7-1, for the flash definition so that all offsets from the first word are abstracted.

Example 7-1 flashType structure

```
typedef int32 flFlash_WriteProc(char *address, unsigned32 data, char *flash);
typedef int32 flFlash_WriteBlockProc(char *address, unsigned32 *data,
                                     unsigned32 size, char *flash);
typedef int32 flFlash_ReadProc(char *address, unsigned32 *value);
typedef int32 flFlash_ReadBlockProc(char *address, unsigned32 *data,
                                    unsigned32 size);
typedef int32 flFlash_EraseProc(char *address, unsigned32 size, char *flash);
typedef int32 flFlash_InitProc(char *address, int32 *flash);
typedef int32 flFlash_CloseProc(char *address, int32 *flash);

typedef struct flashType
{
    char *base;                /* Base Address of flash */
    char *physicalBase;
    unsigned32 size;           /* Size of flash, in bytes          */
    unsigned32 type;           /* Atmel / Intel (CFI) / Unknown    */
    unsigned32 writeSize;      /* Size of physical block           */
    unsigned32 eraseSize;      /* Size of block erase              */
    unsigned32 logicalSize;    /* Size of logical block            */
    flFlash_WriteProc *write;  /* Write one word                   */
    flFlash_WriteBlockProc *writeBlock;  /* Write a block of writeSize bytes */
    flFlash_ReadProc *read;    /* Read one word                    */
    flFlash_ReadBlockProc *readBlock;  /* Read a block of writeSize bytes  */
    flFlash_EraseProc *erase;  /* Erase a block of eraseSize bytes */
    flFlash_InitProc *init;    /* Lock a flash device              */
    flFlash_CloseProc *close;  /* Unlock a flash device            */
    char info[64];             /* Null terminated Info string      */
    struct flashType *next;    /* Pointer to next flash device     */
}
tFlash;
```

7.3.4 Flash types

The library supports the following flash types:

Intel For specifications and general information on the Intel 28Fxxx parts that use the common flash interface, you should contact Intel or visit the Intel web site.

Atmel For specifications and general information on the ATMEL AT29 flash devices that use a different protocol, you should contact Atmel or visit the Atmel web site.

7.3.5 Image management

At the end of the last block of an image, the flash management program writes a data record, or *footer*, that contains information about the image, such as name, start location, and checksum. If the footer cannot fit into the last block of the image, it must be written at the end of the next block (the block fields update accordingly). If this footer is not written, the image is not visible to the boot switcher, and it will not be visible when the flash management program is next run.

Footer structure

The footer structure is a five-word device that contains a pointer to a more detailed structure that, if required, defines the image.

Table 7-2 shows the format for the footer.

The image base address is the start of the first block containing data for this image. If the image is less than one logical block in length, this pointer will be set to the start of the current block.

The library defines a C structure for the footer so that all offsets from the first word are abstracted. Example 7-2 on page 7-9 shows this structure.

Table 7-2 Footer format

Field	Size (in bytes)	Value/usage
Image information base	4	Pointer to the full image descriptor structure.
Image base address	4	Location in flash memory where the image starts.
Signature	4	`0xA0FFFF9F` is an illegal instruction in the ARM instruction set. It can never be produced by compilers so it is a safe value for a unique signature.
Image type	4	Indicates an ARM executable image, SIB, or custom code.
Checksum	4	Checksum for this footer. The checksum is the word sum and is stored as the inverse of the sum.

Example 7-2 FooterType structure

```
typedef struct FooterType {
    void         *infoBase ;  /* Flash Address of any stripped header */
    char         *blockBase ; /* Flash Address of any stripped header */
    unsigned int signature    /* 'Magic' number to prove it's a footer */
    unsigned int type ;       /* Specific Image type ARM,SIB etc */
    unsigned int checksum ;   /* Checksum of this structure only */
} tFooter ;
```

ImageInfo structure

Because the library supports different program image formats, the actual flash programming is separate from image loading. Table 7-3 describes the C structure that holds information about the image.

Table 7-3 ImageInfo structure

Field	Size (bytes)	Value/usage
Image boot flags	4	The boot requirements for the image: Bit 0: NOBOOT. If set, this image is not bootable. The boot switcher ignores it when selecting an image to run. Bit 1: COMPRESSED. If set, this image must be decompressed before being copied into memory. Bit 2: Initialize memory (and MMU) before executing the image. Bit 3: Copy the image into memory before executing it. Bit 4: File system image.
Unique image number	4	Number defined to allow fast searches for the image and easy execution. This is a logical image number and is not related to the order of the images in flash.
Image load address	4	Location in memory where the image must be loaded for execution, if relevant.
Image length	4	Length of image in memory, in bytes, excluding any file header.
Image execute address	4	Execution address of the image in memory.
Image name	16	Name of the image as a 16-byte, null-terminated string.

Table 7-3 ImageInfo structure (continued)

Field	Size (bytes)	Value/usage
Header length	4	Length of any separated header stored with the image.
Header type	4	Type of file: ELF, AIF, binary, or S-record.
Image checksum	4	Checksum to include full image, header, and this image information block.The checksum is the word sum and is stored as the inverse of the sum.

This structure replicates much of the information contained in the header of file formats such as AIF or *Executable and Linkable Format* (ELF) in a form that is accessible to the file-independent routines.

The image data structure contains information about any file header stored in the image space to allow reconstruction of the file, if required.

The image information block is situated immediately after the full image, and any header information is stripped from the input file and stored with the image. The checksum is calculated from the full image, any header information, and the image information block. Example 7-3 shows the `ImageInfoType` structure.

Example 7-3 ImageInfoType structure

```
typedef struct ImageInfoType
{
    unsigned32    bootFlags ;        /* Boot flags, compression etc. */
    unsigned32    imageNumber ;      /* Unique number, selects for boot etc. */
    char          *loadAddress ;     /* Address program should be loaded to */
    unsigned32    length ;           /* Actual size of image */
    PFN           address ;          /* Image is executed from here */
    char          name[16] ;         /* Null terminated */
    char          *headerBase ;      /* Flash Address of any stripped header */
    unsigned32    header_length;     /* Length of header in memory */
    unsigned32    headerType ;       /* AIF, ELF, S-record etc. */
    unsigned32    checksum ;         /* Image checksum (inc. this struct) */
} tImageInfo ;
```

7.4 Functions listed by type

This section lists the library functions by type. The functions are grouped into four main categories:

- Functions that directly access flash memory are described in *Flash library functions, listed by type* on page 7-11.
- Functions related to low-level file structures are described in *File processing functions, listed by type* on page 7-13.
- Functions related to high-level file access are described in *External interface* on page 7-14.
- Functions related to application-defined nonvolatile storage areas are described in *SIB functions* on page 7-15.

7.4.1 Flash library functions, listed by type

This section list the functions that directly access flash memory, and shows where further information can be found on each function.

Locating flash

It is difficult to locate the flash simply by examining a board because accessing the flash area on one target platform might cause an exception on another platform. The library is linked with code that defines the base of flash memory. This allows common applications, such as the download to flash feature of the ARM debuggers, to work on all supported platforms. If this type of application is linked with the µHAL routines, the platform ID routine allows a lookup table of flash base and flash size to be used.

The flash device structure allows you to handle multiple flash parts in a common manner by the library or an application. If a device has an area that can be locked, it is presented as two logical devices, partitioned into lockable and non-lockable. The library does not provide functions to unlock and lock flash, because this memory is reserved for special functionality. The functions for locating flash are:

- *fLib_FindFlash()* on page 7-16
- *fLib_OpenFlash()* on page 7-16.

Single word access

The smallest unit of access is a single word of 32 bits. If flash parts on a given platform are only on 8-bit or 16-bit data paths, these functions mask all issues of byte order and multiple access. The functions are:

- *fLib_ReadFlash32()* on page 7-17

- *fLib_WriteFlash32()* on page 7-17.

Block access

The library uses the concept of logical blocks to improve access times when handling multiple images in flash. These logical blocks hide any physical block mechanism the actual hardware might use to provide a library of high-level routines. The read and write routines are generic so you do not require knowledge of logical blocks, but these routines must synchronize internally to use logical blocks where possible. Each program image occupies one or more blocks of flash. The functions are:

- *fLib_ReadArea()* on page 7-18
- *fLib_WriteArea()* on page 7-18
- *fLib_DeleteArea()* on page 7-19
- *fLib_GetBlockSize()* on page 7-19.

Images in flash

An application must be able to find an image already programmed in flash memory, and find room for a new image. Also, much of the complexity of footers, checksums, and image numbers can be hidden by wrapper routines that allow an application to simply read, write, or verify an image. These functions use the footer list produced by `fLib_FindFooter()`. The functions are:

- *fLib_ReadImage()* on page 7-20
- *fLib_WriteImage()* on page 7-20
- *fLib_VerifyImage()* on page 7-21
- *fLib_FindImage()* on page 7-21
- *fLib_ExecuteImage()* on page 7-22
- *fLib_DeleteImage()* on page 7-22
- *fLib_ChecksumImage()* on page 7-23
- *fLib_ChecksumFooter()* on page 7-23
- *fLib_GetEmptyFlash()* on page 7-24
- *fLib_GetEmptyArea()* on page 7-24.

Image footers

The flash library provides functions to locate, read, build, and write these footers. After flash is scanned for footers with `fLib_FindFooter()`, reading any footer is not merely a case of accessing the returned list, because this would open the application to the actual physical organization and layout of the flash hardware. The image footer functions are:

- *fLib_initFooter()* on page 7-25

- *fLib_ReadFooter()* on page 7-25
- *fLib_WriteFooter()* on page 7-26
- *fLib_VerifyFooter()* on page 7-27
- *fLib_FindFooter()* on page 7-27
- *fLib_BuildFooter()* on page 7-28.

7.4.2 File processing functions, listed by type

The flash library separates file read/write from flash programming. This allows the library to support multiple file formats simply and easily. File formats ELF, AIF, binary, and Motorola S-record are supported.

This section lists, and describes, the file processing functions by type, and shows where further information can be found on each function.

The choice of basic file access or formatted file access depends on the information extracted from the header. When the file header is parsed, `fLib_ReadFileHead()` sets `image->readFile()`, `image->writeFile()` and `image->footer.fileType` appropriately.

If the file does not require any conversion, `image->readFile()` points at `flib_ReadFileRaw()`, and `image->writeFile()` points at `flib_WriteFileRaw()`. Otherwise, such as for an S-record file, the appropriate routine addresses are set in `image->readFile()` and `image->writeFile()`. If a file format has no header, `readFile()` must parse the size bytes of data read from the file from `image->head` first.

Simple file access

The interface to access files on the host is as simple as possible. The file must be opened before access is possible, and must be closed when done. Data is read and written using functions that access the image structure to determine if any format conversion is required (the raw functions are also available).

The simple file access functions are:

- *fLib_ReadFileRaw()* on page 7-29
- *fLib_WriteFileRaw()* on page 7-29
- *fLib_OpenFile()* on page 7-30
- *fLib_CloseFile()* on page 7-30.

File headers and formats

The flash library can maintain an original file format header as part of the image. The data is stored in flash using the following layout, with the original header immediately following the executable image, as shown in Figure 7-1 on page 7-2.

The file inputs must be checked for file type, and stored with respect to the header and code image information. There are two functions to handle parsing of the header information to and from the flash image space:

- *fLib_ReadFileHead()* on page 7-31
- *fLib_WriteFileHead()* on page 7-32.

Formatted file access

Data is accessed using functions that use the image structure for any required format conversion. These functions are:

- *fLib_ReadFile()* on page 7-32
- *fLib_WriteFile()* on page 7-33.

7.4.3 External interface

The flash library separates file read/write from flash programming. This allows the library to support multiple file formats. The supported file formats are ELF, AIF, binary, and Motorola S-record.

This section lists the file processing functions by type, and shows where further information can be found on each function.

External file translation interface

Some external file types, including Motorola S-record and Intel hex, require each input element to be converted. To allow easy access for filter and conversion routines, a small interface has been included.The interface is defined in a C structure, as shown in Example 7-4.

Example 7-4 External file translation interface structure

```
typedef struct
{
    char in_buff[80];  /* Buffer for the ASCII input processing */
    char out_buff[80]; /* Buffer for the processed binary image */
    char * address;    /* Address Image buffer should go to     */
```



```
    int rec_length;    /* Actual size of image buffer          */
    int records;       /* Internal counter for block passage   */
} tProcess_type ;
```

Parameters are in the format shown in Table 7-4.

Table 7-4 Parameters

Field	Size (in bytes)	Value/usage
`in_buff`	80	Input line buffer for the ASCII element read in from an external file.
`out_buff`	80	Storage space for conversion output.
`address`	4	Address for storage, taken from the element header.
`rec_length`	4	True element size taken from the element being processed.
`records`	4	Internal counter for the process type to show the number of elements processed.

The external processing function can be called on a line-by-line basis, and gives the correct data and storage address back to the input function.

There are no individual external file translation interface routines.

7.4.4 SIB functions

Applications sometimes need small amounts of nonvolatile storage. The boot monitor, for example, requires a small block of data to identify which image to run. These small blocks of application-specific information are provided as *System Information Blocks* (SIB).

The following functions are available to create and access SIBs:

- *SIB_Open()* on page 7-36
- *SIB_Close()* on page 7-37
- *SIB_GetPointer()* on page 7-37
- *SIB_Copy()* on page 7-38
- *SIB_Program()* on page 7-38
- *SIB_GetSize()* on page 7-39
- *SIB_Verify()* on page 7-39
- *SIB_Erase()* on page 7-40.

7.5 Flash library functions

This section documents the functions in the flash library. The functions are listed in the order as documented in *Flash library functions, listed by type* on page 7-11. All functions and type definitions are contained in `flash_lib.h`.

7.5.1 fLib_FindFlash()

This function locates the flash memory devices on this platform. If there is more than one device in the system, the application must build a linked list of devices before calling `fLib_OpenFlash()`.

Note

This routine is board-dependent. It might simply return a predefined value, or it might actually scan the memory address space looking for valid flash.

Syntax

```
unsigned int fLib_FindFlash(tFlash **tf)
```

where:

tf is a pointer to an integer variable that will be set to the address of the `flashType` device structure in the system. The routine uses the external array `flash_setup[]`.

Return value

Returns one of the following:

count If one or more flash devices is found, the number of devices is returned. `*tf` is set to the address of the first element of the array of device structures.

0 If no flash is found.

7.5.2 fLib_OpenFlash()

This function initializes the flash device structures for this platform. The `flashmem` array `Default_Flash_Setup` in `FL_flash_proc.c` holds default flash functions for Atmel and Intel flash devices. The values in the appropriate structure are used to initialize the device structure.

Syntax

```
int fLib_OpenFlash(tFlash *flashmem)
```

where:

flashmem is a pointer to the first flash memory information structure.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.3 fLib_ReadFlash32()

This function calls the `read()` function from the *flashmem* structure and reads one 32-bit word from the flash at the given address.

Syntax

```
int fLib_ReadFlash32(unsigned int *address, unsigned int *value,
                     tFlash *flashmem)
```

where:

address is a pointer to the address of the flash memory to be read.

value is a pointer to the memory address where the flash should be copied.

flashmem is a pointer to the flash device structure to allow access to the flash read/write routines.

Return value

Returns one of the following:

0 If successful. The memory at *value* now holds the results.

-1 If not successful.

7.5.4 fLib_WriteFlash32()

This function writes one 32-bit word to the flash at the given address.

Syntax

```
int fLib_WriteFlash32(unsigned int *address, unsigned int value,
                      tFlash *flashmem)
```

where:

address	is a pointer to the address of the flash memory to be written to.
value	is the data to be written to the specified flash address.
flashmem	is a pointer to the flash device structure to allow access to the flash read/write routines.

Return value

Returns one of the following:

0	If successful.
-1	If not successful.

7.5.5 fLib_ReadArea()

This function reads an area of *size* bytes from flash memory.

Syntax

```
int fLib_ReadArea(unsigned int *address, unsigned int *data,
                  unsigned int size, tFlash *flashmem)
```

where:

address	is a pointer to the address of the flash memory to be read.
data	is a pointer to the location the data is to be copied to.
size	is the size, in bytes, of the data area.
flashmem	is a pointer to the flash device structure to allow access to the flash read/write routines.

Return value

Returns one of the following:

0	If successful.
-1	If not successful.

7.5.6 fLib_WriteArea()

This function writes an area of *size* bytes to flash memory.

Syntax

```
int fLib_WriteArea(unsigned int *address, unsigned int *data,
                   unsigned int size, tFlash *flashmem)
```

where:

address	is a pointer to the address of the flash memory to be written.
data	is a pointer to the data to be written.
size	is the size, in bytes, of the data area.
flashmem	is a pointer to the flash device structure to allow access to the flash read/write routines.

Return value

Returns one of the following:

0	If successful.
-1	If not successful.

7.5.7 fLib_DeleteArea()

This function deletes (erases) an area of flash memory.

Syntax

```
int fLib_DeleteArea(unsigned int *address, unsigned int size,
                    tFlash *flashmem)
```

where:

address	is a pointer to the address of the flash memory to be deleted.
size	is the size, in bytes, of the data area to be erased.
flashmem	is a pointer to the flash device structure to allow access to the flash read/write routines.

Return value

Returns one of the following:

0	If successful.
-1	If not successful.

7.5.8 fLib_GetBlockSize()

This function returns the size, in bytes, of the logical block for this platform.

Note

These logical blocks cannot be smaller than the largest device physical block size. This block size will be a multiple of the erase block size.

Syntax

```
unsigned int fLib_GetBlockSize(tFlash *flashmem)
```

where:

flashmem is a pointer to the flash device structure to return the size.

Return value

Returns one of the following:

size If the flash block size can be determined, the size of the block is returned.

0 If the size cannot be determined.

7.5.9 fLib_ReadImage()

This function reads the image from flash memory as defined in *image*->footer. The specified *image*->ramBase pointer cannot be NULL.

Syntax

```
int fLib_ReadImage(tFooter *foot, tFlash *flashmem)
```

where:

foot is a pointer to the footer structure defining the image pointer for the image to be read.

flashmem is a pointer to the flash device structure to allow access to the flash read/write routines.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.10 fLib_WriteImage()

This function writes the image selected by the specified image structure. The image structure must be fully defined.

Syntax

```
int fLib_WriteImage(tFooter *foot, tFlash *flashmem)
```


where:

foot is a pointer to the footer structure defining the image pointer for the image to be copied.

flashmem is a pointer to the flash device structure to allow access to the flash read/write routines.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.11 fLib_VerifyImage()

This function verifies that the image, selected by the specified image structure, matches the image as programmed. The image structure must be fully defined.

Syntax

```
int fLib_VerifyImage(tFooter *foot)
```

where:

foot is a pointer to the footer structure defining the image pointer for the image to be verified.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.12 fLib_FindImage()

This function scans the list of flash footers looking for a footer with an image number that matches the specified number. If the specified footer pointer is not NULL, the footer is copied from flash.

Syntax

```
int fLib_FindImage(tFooter **list, unsigned int imageNo,
                   tFooter *foot)
```

where:

list	is a pointer to a list of pointers to footers.
imageNo	is the unique number of the image to be located.
foot	is a pointer to the location where the found footer should be copied.

Return value

Returns one of the following:

0	If successful.
-1	If not successful.

7.5.13 fLib_ExecuteImage()

This function executes the image selected by the specified image footer.

Syntax

```
int fLib_ExecuteImage(tFooter *foot)
```

where:

foot	is a pointer to the footer that defines the image to be executed.

Return value

Returns one of the following:

No return	If successful, the function does not return.
-1	If not successful.

7.5.14 fLib_DeleteImage()

This function deletes the image in flash selected by the specified image footer.

Syntax

```
int fLib_DeleteImage(tFooter *foot)
```

where:

foot	is a pointer to the footer that defines the image to be deleted.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.15 fLib_ChecksumImage()

This function calculates the checksum for the specified image. The image structure and associated footer must be fully defined, but the contents of the image structure are not summed. The checksum is a word sum, and is inverted before being stored.

Syntax

```
int fLib_ChecksumImage(tFooter *foot, unsigned int *sum)
```

where:

foot is a pointer to the footer structure defining the image pointer for the image to be check-summed.

sum is a pointer to location where the image checksum is to be stored.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.16 fLib_ChecksumFooter()

This function calculates the checksum for the specified image. The image structure and associated footer must be fully defined, but the contents of the image structure are not summed. If the image sum value is -1, only the footer value will be calculated. The checksums are word sums, and are inverted before being stored.

Syntax

```
int fLib_ChecksumFooter(tFooter *foot, unsigned int *foot_sum,
                        unsigned int *image_sum)
```

where:

foot is a pointer to the footer structure for the footer and image to be check-summed.

foot_sum is a pointer to the location where the footer checksum is to be stored.

image_sum is a pointer to the location where the image checksum is to be stored.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.17 fLib_GetEmptyFlash()

This function scans the list of flash footers, looking for an empty area from *start*, of at least *unused* size.

Syntax

```
int fLib_GetEmptyFlash(tFooter **list, unsigned int *start,
                       unsigned int *location,
                       unsigned int empty, Flash *flash)
```

where:

list is a pointer to a list of pointers to footers.

start is a pointer to the start location required in flash memory.

location is a pointer to the start of the flash area capable of housing the image.

empty is the size of the empty area required in flash memory.

flash is a pointer to the location from where the footer image is to be copied.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.18 fLib_GetEmptyArea()

This function scans flash footers, looking for any empty area of at least *empty* size.

Syntax

```
int fLib_GetEmptyArea(tFooter **list, unsigned int empty,
                      tFlash *flash)
```

where:

list	is a pointer to a list of pointers to footers.
empty	is the size of the empty area required in flash memory.
flash	is a pointer to the location from where the footer image is to be copied.

Return value

Returns one of the following:

0	If successful.
-1	If not successful.

7.5.19 fLib_initFooter()

This function initializes the footer at *foot* with known values (-1, or 0xFFFFFFFF). This sets up the footer to a known state. The value -1 is the general value of unprogrammed flash.

Syntax

```
int fLib_initFooter(tFooter *foot, int ImageSize, int type)
```

where:

foot	is a pointer to the footer structure for initialization.
ImageSize	is the size of the image, if known.
type	is a footer type such as an ARM executable or SIB (bit patterns defined in flash_lib.h).

Return value

Returns one of the following:

0	If successful.
-1	If not successful.

7.5.20 fLib_ReadFooter()

This function reads the footer at *start* in flash memory to *foot* in memory.

Syntax

```
int fLib_ReadFooter(unsigned int *start, tFooter *foot,
                    tFlash *flash)
```

where:

start	is a pointer to the location of the footer image in flash memory.
foot	is a pointer to the location the footer image is to be copied to.
flash	is a flash device structure for access to flash access routines.

Return value

Returns one of the following:

0	If successful.
-1	If not successful.

7.5.21 fLib_WriteFooter()

This function writes a footer to flash memory. *image_data* contains the complete image footer to be written, including the checksum. Because the footer contains a pointer to the end of the flash block, this function uses the pointer to determine where the footer should be written.

Syntax

```
int fLib_WriteFooter(tFooter *foot, tFlash *flash,
                     unsigned int *foot_data,
                     unsigned int *image_data)
```

where:

foot is a pointer to the location where the footer image is to be copied.

flash is a structure with pointers to flash access routines.

foot_data is the location of footer data in RAM to be copied to a flash location.

image_data is a pointer to the location of the image information to be copied to flash.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.22 fLib_VerifyFooter()

This function verifies the footer at *foot*. It checks the signature word, and also checks that the checksum is correct.

Syntax

```
int fLib_VerifyFooter(tFooter *foot, tFlash *flash)
```

where:

foot is a pointer to the footer image to be verified.

flash is a structure with pointers to flash access routines.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.5.23 fLib_FindFooter()

This function scans the flash memory from *start* for *size* bytes, returning a list of pointers to the image footers. The pointer *list* should point to an area of RAM supplied by the application, and should be large enough to contain a pointer to each logical block of flash in the specified area.

If the size is defined as zero, only the address of the next footer found is returned.

Syntax

```
unsigned int fLib_FindFooter(unsigned int *start,
                  unsigned int size, tFooter *list[],
                  tFlash *flash)
```

where:

start is a pointer to the address of the flash memory to be scanned.

size is the size, in bytes, of the flash memory.

list is a pointer to a list of pointers to footers.

flash is a structure with pointers to flash access routines.

Return value

Returns the number of flash footers found.

7.5.24 fLib_BuildFooter()

This function builds a footer for the specified image. The image structure already contains all information about the program image in memory. This function must convert these pointers to their final values in flash.

Syntax

```
int fLib_BuildFooter(tFooter *foot, tFlash *flash)
```

where:

foot is a pointer to the footer to be built.

flash is a structure with pointers to flash access routines.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.6 File processing functions

This section documents the set of file processing function calls. The functions are listed in the order as documented in *File processing functions, listed by type* on page 7-13. All functions and type definitions are contained in `flash_lib.h`.

7.6.1 fLib_ReadFileRaw()

This function reads up to *`size`* bytes from the open file *`fp`*.

Syntax

```
unsigned int fLib_ReadFileRaw(unsigned int *value,
                    unsigned int size,
                    tFile_IO *file_IO, tFILE *fp)
```

where:

`value`	is a pointer to the memory address where the contents of the file is copied.
`size`	is the number of bytes to be read.
`file_IO`	is a pointer to a structure that accesses the external file input/output by way of simple input/output routines.
`fp`	is a pointer to an open file stream from which to read file data.

Return value

Returns one of the following:

count	If successful, the number of bytes read is returned.
0	If not successful.

7.6.2 fLib_WriteFileRaw()

This function writes up to *`size`* bytes to the open file *`fp`*.

Syntax

```
unsigned int fLib_WriteFileRaw(unsigned int *value,
                    unsigned int size,
                    tFile_IO *file_IO, tFILE *fp)
```

where:

value is a pointer to the memory address from where the contents of the file is copied.

size is the number of bytes to be written.

file_IO is a pointer to a structure that accesses the external file by way of simple input/output routines.

fp is a pointer to an open file stream to which file data will be written.

Return value

Returns one of the following:

count If successful, the number of bytes written is returned.

0 If not successful.

7.6.3 fLib_OpenFile()

This function opens a file of the given *filename* in the given *mode*.

Syntax

```
File *fLib_OpenFile(char *filename, char *mode,
tFile_IO *file_IO)
```

where:

filename is a pointer to the name of the file on the host.

mode is the mode in which the file should be opened, such as `rb` for read-only.

file_IO is a pointer to a structure that accesses the external file input/output by way of simple input/output routines.

Return value

Returns one of the following:

pointer If successful, a pointer to the file on the host is returned.

0 If not successful.

7.6.4 fLib_CloseFile()

This function closes the specified file on the host.

Syntax

```
int fLib_CloseFile(File *file, tFile_Io *file_IO)
```

where:

file is a pointer to the file on the host.

file_IO is a pointer to a structure that accesses the external file input/output by simple input/output routines.

Return value

Returns one of the following:

0 If successful.

-1 If not successful.

7.6.5 fLib_ReadFileHead()

This function reads the file header, determines the file type, and sets fields in *image* from the data. The number of bytes read is returned in the field pointed to by *size*. The header is copied to the buffer already defined in *image*->head.

Syntax

```
unsigned int fLib_ReadFileHead(File *file, tImageInfo *image,
                    unsigned int *size, tFile_IO *file_IO)
```

where:

file is a pointer to the file on the host.

image is a pointer to the image structure.

size is a pointer to size of the data read from the host.

file_IO is a pointer to a structure that accesses the external file input/output by simple input/output routines.

Return value

Returns one of the following:

filetype If the file type is known, the file type is returned as an **`unsigned int`** `fileType` (`ENUM_FILETYPE`).

0 If the file type is unknown.

7.6.6 fLib_WriteFileHead()

This function writes the header pointed to by the *`image`*`->footer` to the specified file. The header is parsed and the `writeFile` routine pointer is updated.

Syntax

```
unsigned int fLib_WriteFileHead(File *file, tImageInfo *image,
                      tFile_IO *file_IO)
```

where:

`file` is a pointer to the file on the host.

`image` is a pointer to the image structure.

`file_IO` is a pointer to a structure that accesses the external file input/output by way of simple input/output routines.

Return value

Returns one of the following:

count If successful, the number of bytes written is returned.

0 If there is no header.

7.6.7 fLib_ReadFile()

This function reads (and converts), from the open file, words of up to 32 bits.

Syntax

```
unsigned int fLib_ReadFile(unsigned int *value,
                    unsigned int size, tImageInfo *image,
                    tFile_IO *file_IO)
```


where:

value	is a pointer to the memory address where the file data is copied.
size	is the number of bytes to be read.
image	is a pointer to the image structure.
file_IO	is a pointer to a structure that accesses the external file input/output by way of simple input/output routines.

Return value

Returns one of the following:

count	If successful, the number of bytes read is returned.
0	If not successful.

7.6.8 fLib_WriteFile()

This function converts and writes, from the open file, words of up to 32 bits.

Syntax

```
unsigned int fLib_WriteFile(unsigned int *value,
                    unsigned int size, tImage *image,
                    tFile_IO *file_IO)
```

where:

value	is a pointer to the memory address.
size	is the number of bytes to be written.
image	is a pointer to the image structure.
file_IO	is a pointer to a structure that accesses the external file input/output by way of simple input/output routines.

Return value

Returns one of the following:

count	If successful, the number of bytes written is returned.
0	If not successful.

7.7 SIB functions

Applications sometimes need small amounts of non-volatile storage. The boot monitor, for example, requires a small block of data to identify which image to run. These small blocks of application-specific information are provided as *System Information Blocks* (SIB).

The flash library can create a large block of memory called a SIB flash block that can then be used by various applications to create or access individual SIBs within the larger block. It is also possible for an application to ask for an entire SIB flash block if, for example, the application requires very large SIBs or if the SIBs must not be accidently modified by another applications.

The following functions are available to create and access SIBs:

- *SIB_Open()* on page 7-36
- *SIB_Close()* on page 7-37
- *SIB_GetPointer()* on page 7-37
- *SIB_Copy()* on page 7-38
- *SIB_Program()* on page 7-38
- *SIB_GetSize()* on page 7-39
- *SIB_Verify()* on page 7-39
- *SIB_Erase()* on page 7-40.

7.7.1 The SIB flash block

The SIB flash block contains multiple SIBs as shown in Figure 7-2 on page 7-35. The SIBs contained within the SIB flash block are application-dependent. The flash library defines how the SIB blocks are accessed but does not define the contents of the individual SIBs.

The size limit for SIB blocks is 512 bytes and the index limit (number of SIBs with the same name) is 64.

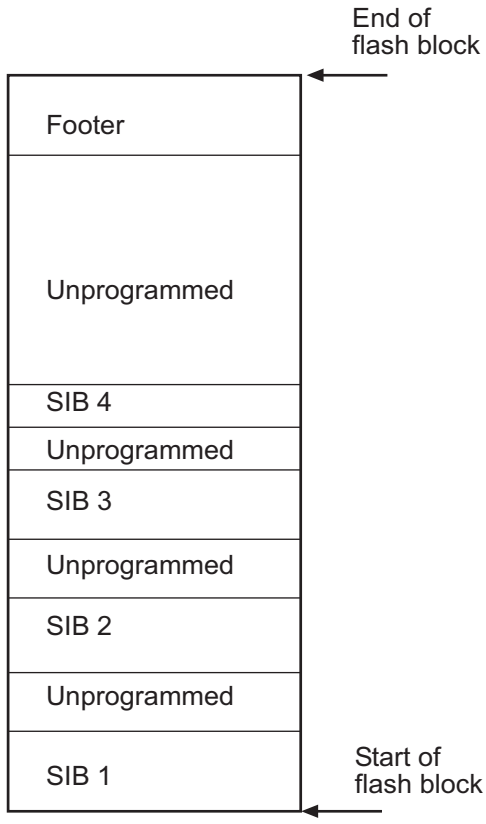

Figure 7-2 SIB flash block

Flash blocks containing SIBs must be identifiable. The SIB flash block footer contains a word that identifies the block as a SIB flash block. A SIB information block precedes the footer and contains additional information about the block. Table 7-5 describes the contents of the SIBInfo structure.

Table 7-5 SIBInfo structure

Field	Size (in bytes)	Value/usage
SIB unique number	4	Unique number for the SIB flash block (or blocks) for system reference.
SIB block extension	4	Pointer to the start of this SIB flash block (some SIBs require more than one flash block).
Label	16	Text label for identification of the SIB flash block. This will generally be the initializing system name.
Checksum	4	Checksum for this footer. The checksum is the word sum and is stored as the inverse of the sum.

The SIB is defined as a C structure, as shown in Example 7-5.

Example 7-5 SIBInfoType structure

```
typedef struct SIBInfoType
{
  unsigned32 SIB_number;    /* Unique number of SIB Block */
  unsigned32 SIB_Extension; /* Base of SIB flash block */
  char    Label[16];     /* String space for ownership string */
  unsigned32 checksum;   /* SIB Image checksum */

}tSIBInfo;
```

7.7.2 SIB_Open()

`SIB_Open()` scans flash for SIB blocks and indexes the SIBs in a linked list for faster access (an application might have multiple SIBs in different blocks). The application can then access the SIBs by their index. This routine uses the `fLib_FindFlash()`, `fLib_OpenFlash()`, and `fLib_FindFooter()` functions.

Syntax

```
int SIB_Open(char *idString, int *sibCount, int privFlag)
```

where:

idString is provided by the application and is an identification string that will be used to locate existing blocks and mark new ones.

sibCount is set to the number of SIBs found.

privFlag is zero for common access or non-zero for private access. Private access means that the entire flash block is private.

Return value

Returns one of the following:

-1 If *idString* is already set.

0 If successful.

7.7.3 SIB_Close()

`SIB_Close()` frees SIB access.

Syntax

```
int SIB_Close(char *idString)
```

where:

idString is provided by the application and is an identification string that will be used to locate existing blocks and mark new ones.

Return value

Returns one of the following:

-1 If *idString* is already set.

0 If successful.

7.7.4 SIB_GetPointer()

`SIB_GetPoiner()` gets the start address of SIB user data.

Syntax

```
int SIB_GetPointer(int sibIndex, void **dataBlock)
```

where:

sibIndex is the index number of the SIB. The SIB indexes were identified by `SIB_Open()`.

dataBlock is set to the address if the SIB user data.

Return value

Returns one of the following:

-1 If not successful.

0 If successful. *dataBlock* is set to the address.

7.7.5 SIB_Copy()

`SIB_Copy()` gets a local copy of the user data in a SIB.

Syntax

```
int SIB_Copy(int sibIndex, void *dataBlock, int dataSize)
```

where:

sibIndex is the index number of the SIB. The SIB indexes were identified by `SIB_Open()`.

dataBlock is the base source address.

dataSize is the free space at the source address.

Return value

Returns one of the following:

-1 If not successful.

0 If successful.

7.7.6 SIB_Program()

`SIB_Program()` creates a new SIB or updates an existing SIB with new user data.

Syntax

```
int SIB_Program(int sibIndex, void *dataBlock, int dataSize)
```


where:

sibIndex is the new index number of the SIB. The existing SIB indexes were identified by `SIB_Open()`.

dataBlock is the base source address.

dataSize is the free space at the source address.

Return value

Returns one of the following:

-1 If not successful.

0 If successful.

7.7.7 SIB_GetSize()

`SIB_GetSize()` gets the size of SIB data.

Syntax

```
int SIB_GetSize(int sibIndex, int *dataSize)
```

where:

sibIndex is the index number of the SIB. The SIB indexes were identified by `SIB_Open()`.

dataSize is set to the size of the SIB.

Return value

Returns one of the following:

-1 If not successful.

0 If successful.

7.7.8 SIB_Verify()

`SIB_Verify()` verifies the SIB is intact by checking the signature and checksum.

Syntax

```
int SIB_Verify(int sibIndex)
```

where:

sibIndex is the index number of the SIB. The SIB indexes were identified by `SIB_Open()`.

Return value

Returns one of the following:

-1 If not successful.

0 If successful.

7.7.9 SIB_Erase()

`SIB_Erase()` erases the SIB.

Syntax

```
int SIB_Erase(int sibIndex)
```

where:

sibIndex is the index number of the SIB. The SIB indexes were identified by `SIB_Open()`. The entry in `active_sibs[sibIndex]` is set to NULL.

Return value

Returns one of the following:

-1 If not successful.

0 If successful.

7.8 Using the library

The flash library provides a wide range of routines, so it is recommended that you understand how they work together. The sequences described in this section do not give specific constructs, however they give a general indication of usage.

7.8.1 Starting up and finding flash

When the programming application starts on the target, the application must:

1. Locate the flash.
2. Verify that it is supported.
3. Scan for any images that have already been programmed.

Example 7-6 shows how these actions are performed.

Example 7-6 Starting up

```
unsigned int fLib_FindFlash(tFlash **tf);
int fLib_OpenFlash(tFlash *flashMem);
unsigned int fLib_FindFooter(unsigned int *start, unsigned int size,
             tFooter **list[], tFlash *flash);
```

7.8.2 Reading a file into memory

It is only necessary to read an image from the host once. Therefore, it is recommended that you do not integrate the file that is read into the programming command. Instead, perform a separate step to read the file first, such as by using a combined read-and-program command. Example 7-7 shows these actions are performed.

Example 7-7 Reading into memory

```
File *fLib_OpenFile(char *filename, char *mode, tFile_IO *file_IO);
unsigned int fLib_ReadFileHead(File *file, tImage *image,
                  unsigned int *size, tFile_IO *file_IO);
unsigned int fLib_ReadFile(unsigned int *value, unsigned int size, tImage *image,
                  tFile_IO *file_IO);
int fLib_CloseFile(File *file, tFile_IO *file_IO);
```

7.8.3 Preparing and programming an image

After the image is loaded into memory, space must be found for the image and the image footer has to be built before the image can be programmed. If an image is relocated into memory when executed, it can be programmed into any available flash space. If there is no room for the desired image in flash, an existing image will have to be deleted.

The image number must be checked to ensure that it is unique. If image numbers were not unique, there would be problems selecting one of the multiple images to execute. After the image is written, the footer should be rescanned to update the image list. (The image numbers are logical numbers and are not related to the order of the images in flash.) Example 7-8 shows how these actions are performed.

Example 7-8 Programming

```
int fLib_FindImageNum(tFooter **list, unsigned int imageNo, tFooter *foot);
int fLib_GetEmptyFlash(tFooter **list, unsigned int *search_start,
                unsigned int &location, unsigned int empty, tFlash *flash);
```

or:

```
int fLib_GetEmptyArea(tFooter **list, unsigned int &location,
                unsigned int empty, tFlash *flash );
int fLib_DeleteArea(unsigned int *address, unsigned int size, tFlash *flash);
int fLib_BuildFooter(tFooter *foot, tFlash *flash);
int fLib_ChecksumFooter(tFooter *footer, unsigned32 *foot_sum,
                unsigned32 *image_sum);
int fLib_WriteImage(tImageInfo *image, tFlash *flash, unsigned32 *current,
                tFooter *foot);
```

or:

```
int fLib_WriteArea(unsigned32 *address, unsigned32 *data,
                unsigned32 size, tFlash *flashmem);
tFooter* fLib_WriteFooter(tFooter *foot, tFlash *flash,
                unsigned32 *foot_data, unsigned32 *image_data);
unsigned int fLib_FindFooter(unsigned int *start,
                unsigned int size, tFooter *list[], tFlash *flash);
```

7.8.4 Reading an image to a file

The process described in *Preparing and programming an image* can be reversed to produce a file on the host from a flash image. Example 7-9 shows how these actions are performed.

Example 7-9 Reading

```
int fLib_FindImage(tFooter **list, unsigned32 imageNo,tFooter *foot);
int fLib_VerifyFooter(tFooter *foot, tFlash *flash );
int fLib_ReadImage(tFooter *foot, tFlash *flash);
int fLib_ChecksumImage(tFooter *footer, unsigned32 *image_sum);
tFILE *fLib_OpenFile(char *filename, char *mode, tFile_IO * file_IO);
              unsigned int fLib_WriteFileHead(tFILE *file, tImageInfo *image,
              tFile_IO * file_IO)
fLib_WriteFile(unsigned32 *value, unsigned32 size,
tImageInfo *image, tFile_IO * file_IO);
int fLib_CloseFile(tFILE *file, tFile_IO * file_IO);
```

7.8.5 Executing an image

This process is very similar to the image read, but instead of copying to memory and then to a file, the image is copied to memory only if specified, and then processor control is passed to the image. Example 7-10 shows how these actions are performed.

Example 7-10 Executing

```
int fLib_FindImage(tFooter **list, unsigned int imageNo, tFooter *foot);
int fLib_VerifyFooter(tFooter *foot, tFlash *flash );
int fLib_ReadImage(tFooter *foot, tFlash *flash);
int fLib_ChecksumImage(tFooter *footer, unsigned32 *image_sum);
int fLib_ExecuteImage(tFooter *foot);
```

7.9 Rebuilding the flash library

Use the project files or makefiles to rebuild the flash library.

7.9.1 PC project files

You can build the flash library with SDT 2.5 project manager files (`.apj`) or ADS 1.0 CodeWarrior project files (`.mcp`).

7.9.2 Unix makefile

The CD has a makefile for use on a Unix workstation.

There is a makefile for rebuilding the flash library for a single development board and processor combination. For example, if you copied `unix/source` contents to `/AFS` use `/AFS/source/Integrator/FlashLibrary/Build/Integrator.b/makefile` to rebuild the library for the Integrator board with an ARM940T processor.

You must maintain the hierarchy of the CD directories when you copy the files from the CD to your workstation. The makefile defines ROOT as the root of the build tree and is needed by `mk`. TOOLS is the tools directory that contains build tools of various kinds.

For general information on makefiles and directory structure, see *AFS source structure* on page 11-4.

Chapter 8
Using the ARM Flash Utilities

This chapter discusses the operation of utilities for accessing flash memory.

The *ARM Flash Utility* (AFU) provides functions for accessing the flash library as described in Chapter 7 *Flash Library Specification*.

The *ARM Boot Flash Utility* (BootFU) provides functions for programming the boot and FPGA areas of flash memory.

This chapter contains the following sections:

See also Chapter 4 *ARM Boot Monitor* and Chapter 7 *Flash Library Specification* for additional information about images in flash memory.

8.1 About the AFU

The AFU is an application for manipulating and storing data within a system that uses the flash library. It is a target-based application designed to allow you to download code onto an ARM development system, maintaining the ARM Flash Library structure. This enables you to use ARM boot systems to run the code on the board.

The AFU can handle the following formats:

- ELF
- relocatable AIF
- fixed AIF
- plain binary
- Motorola S-record format.

The AFU performs the following functions:

1. Reads the files from a host system.
2. Analyzes the required location (if applicable).
3. Writes the code image into the correct location in memory.
4. Strips the file header from the image and stores it immediately after the image. This allows full reconstruction of the file where possible.
5. Adds an image information block after any file header information to allow flash library-aware drivers to identify and run the code segments.
6. Stores the flash footer block at the subsequent block boundary to the image information block. This allows for quick image search routines.

8.2 Starting the AFU

The AFU is designed to run within an ARM debug environment such as the ARM Multi-ICE server and the *ARM Debugger for Windows* (ADW) environment. The target processor must be configured to run without caches. To set up and run the AFU:

1. Start up a debug session for the board requiring flash download and load the image `armfu.axf`.
2. Ensure the console window is active. If it is not, select **Console** from the **View** menu.
3. Run the code by pressing F5, or by typing `go` in the command window, or by clicking on the **GO** icon.

The console window appears in the foreground and becomes active, with the AFU header similar to the following:

```
ARM Firmware Suite
Copyright (c) ARM Ltd 1999-2000. All rights reserved.

ARM Flash Utility
Program Version 1.0
Date: 29 Jan 2000
```

The AFU scans for flash components and default to the device at the lowest address. After the flash device is selected, the AFU scans the flash for any images currently programmed.

User interface is signaled by the AFU prompt `AFU>`. This indicates that you can operate the AFU.

8.3 AFU commands

This section describes each of the command-line entries the AFU can accept. It describes the parameters required by the command, and shows the output generated by the AFU. Table 8-1 lists the AFU commands.

Table 8-1 AFU commands

Command	Short form	Description
List on page 8-5	`L`	Lists image footers
DiagnosticList on page 8-6	`dia`	Examines flash blocks for possible problems
TestBlock on page 8-11	`t`	Tests the integrity of the block
Delete on page 8-12	`delete`	Deletes a full image from flash
DeleteBlock on page 8-12	`delete`	Deletes a specified block
DeleteAll on page 8-13	`deletea`	Erases all flash blocks
Program on page 8-13	`p`	Takes an image from a host computer and places it in flash
Read on page 8-17	`r`	Takes an image from memory and stores it on the host computer
Quit on page 8-17	`q`	Quits the current AFU session
Help on page 8-17	`h`	Displays the AFU command summary
Identify on page 8-18	`i`	Identifies the current active flash device
Swap on page 8-19	`s`	Allows you to change between the different FLS flash devices

8.3.1 User command explanation

The AFU has a very basic command interpreter with parsing for fast command typing. There is no command-line buffering. You have to reenter in full any incorrect command input.

The syntax of the commands shown in *AFU commands* on page 8-4 shows both the full command and the minimum character(s) required for the AFU parser to run the command (denoted by the underscore symbol).

8.3.2 List

When the AFU scans the memory, it creates a list of recognized footers throughout the memory block. The `List` command shows this list, and other information, from the image footers and image information.

Syntax

`List`

Output

The list is formatted as shown in Table 8-2.

Table 8-2 Output format of list

Name	Format	Explanation
`Image`	*n*	The specific image number you must enter when uploading and programming the image. This number will be unique in flash memory.
`Block`	*n*	The start block of the image.
`End block`	*n*	The final block of the image that contains, at least, the five word footer.
`Address`	`0x`*hhhhhhhh*	The address of the start of the image in memory. This address corresponds with the start of the start block.
`Exec`	`0x`*hhhhhhhh*	The execution address of the image in memory. This might not be within the flash memory boundaries if the image is to be copied to another memory location.
`Name`	*text*	The textual name given to the image when programming into memory.

Example

In Example 8-1, the only image in the memory system is a single block image at block 17 called `hello`, where the entry point is at the start of the image.

Example 8-1 List command

```
AFU>List
Image 1 Block 17 End Block 17 address 0x24220000 exec 0x24220000 - name hello
```

8.3.3 DiagnosticList

This command has multiple functions to allow examination of the flash blocks to scan for possible problems. This command will rarely be used in normal operation of the AFU.

Syntax

`diagnosticlist {all|section b`*n*`|footer b`*n*`|dump b`*n*`}`

where:

`all` Scans through every block in the current device. Outputs the usage of each block, as shown in Table 8-3.

Table 8-3 Output format of Diagnostic List All

Name	Format	Explanation
`Block`	*n*	The start block of the image.
`Image number`	*n*	The specific image number you must enter when uploading and programming the image. This number will be unique in flash memory.
`Type`	*n*	The image type value, taken from the footer information.
`Image`	*text*	The textual name given to the image when programming into memory.

If the image spans multiple blocks, each block is listed as used by the same image number and image name.

The AFU can only recognize blocks that have been programmed to conform to the flash library specification (see *Image management* on page 7-8).

The list will only show images that have the correct flash image footer. Any images not conforming to this are not shown, and the blocks occupied by these are marked as unused.

The `DiagnosticList all` command (and the following `DiagnosticList section` command) indents the used blocks to ensure that they can be noticed. This is useful when the list is rapidly scrolling down the console window.

`section b`*n*

Presents a list of identical format to `DiagnosticList All`, but only lists the ten subsequent flash blocks after the user input start block *n*, where *n* is a logical flash block number.

`footer b`*n*

Shows the footers of the subsequent five blocks after the input block number *n*, where *n* is a logical flash block number, formatted to describe the displayed information, as shown in Table 8-4.

Table 8-4 Output format of DiagnosticList footer

Name	Format	Explanation
`Block`	*n*	The start block of the image, where *n* is a logical flash block number.
`Address`	`0x`*hhhhhhhh*	The address of the start of the image in memory. This address will correspond with the start of the start block.
`infoBase`	`0x`*hhhhhhhh*	The address of the image information block in memory, that will be at the end of the image.
`blockBase`	`0x`*hhhhhhhh*	The address of the start of the image in flash memory.
`Signature`	`0x`*hhhhhhhh*	A unique word value to distinguish the footer from any code to allow for faster search operations.
`Type`	`0x`*hhhhhhhh*	The image type as defined in the file `flash_lib.h`.
`Checksum`	`0x`*hhhhhhhh*	The logical inverse of the word sum of the preceding four words

The `DiagnosticList Footer` command does not only list valid footer information, but it also lists any data found in the footer area of the listed blocks. You must analyze the data given to see the valid footers, the areas of code, and the unused blocks.

`dump b`*n* Produces a hexadecimal dump of the first four words of the ten blocks following the input block number *n* (where *n* is a logical flash block number), as shown in Table 8-5. The `DiagnosticList Dump` command makes block-based (not image-based) selections, and displays the first four words of each block in the selected area starting at the input block.

Table 8-5 Output format of DiagnosticList dump

Name	Format	Explanation
`Block`	*n*	The block number for the data being listed.
`Address`	`0x`*hhhhhhhh*	The address of the first word being listed, that will correspond with the block number shown.

Examples

The following examples (Example 8-2 to Example 8-5 on page 8-11) demonstrate usage of each `DiagnosticList` command.

Example 8-2 DiagnosticList all command

```
AFU>DiagnosticList All
Block Number 0 unused
 Block 1 Image Number 1 type 1 Used by image hello_world
 Block 2 Image Number 2 type 1 Used by image dhrystone
 Block 3 Image Number 2 type 1 Used by image dhrystone
Block Number 4 unused
Block Number 5 unused
Block Number 6 unused
Block Number 7 unused
.
.
.
Block Number 255 unused
```


Example 8-3 DiagnosticList footer command

```
AFU>DiagnosticList Footer B1
Footer for Block   1 at Address 0x24020000
infoBase  : 0x2402330c
blockBase : 0x24020000
signature : 0xa00fff9f
type      : 0x00000001
checksum  : 0x0beddde0

Footer for Block   2 at Address 0x24040000
infoBase  : 0x0a0000f0
blockBase : 0xe3570078
signature : 0x0a0000ae
type      : 0x00000001
checksum  : 0xe5940000

Footer for Block   3 at Address 0x24060000
infoBase  : 0x2404330c
blockBase : 0x2404f000
signature : 0xa00fff9f
type      : 0x00000001
checksum  : 0x0be5fde0

Footer for Block   4 at Address 0x24080000
infoBase  : 0xffffffff
blockBase : 0xffffffff
signature : 0xffffffff
type      : 0xffffffff
checksum  : 0xffffffff

Footer for Block   5 at Address 0x240a0000
infoBase  : 0xffffffff
blockBase : 0xffffffff
signature : 0xffffffff
type      : 0xffffffff
checksum  : 0xffffffff
```

where:

`Block 1` Is a correct footer because the signature is valid, and the `infoBase` and `BlockBase` are within the bounds of the image address (similar to the block address).

`Block 2` Is either some random code of an image that spans two blocks (in this case), or a block that does not conform to the library specification.

`Block 3` Is the footer for block 2 and block 3 (`blockBase` shows the start of the image).

`Block 4` Is unused.

`Block 5` Is unused.

Example 8-4 DiagnosticList dump command

```
AFU>DiagnosticList dump B1
Block  1
Address 0x24020000  :  0xe59f0034  0xe59f1034  0xe59f3034  0xe1500001

Block  2
Address 0x24040000  :  0xffffffff  0xffffffff  0xffffffff  0xffffffff

Block  3
Address 0x24060000  :  0xe59f0034  0xe59f1034  0xe59f3034  0xe1500001

Block  4
Address 0x24080000  :  0xffffffff  0xffffffff  0xffffffff  0xffffffff

Block  5
Address 0x240a0000  :  0xffffffff  0xffffffff  0xffffffff  0xffffffff

Block  6
Address 0x240c0000  :  0xffffffff  0xffffffff  0xffffffff  0xffffffff

Block  7
Address 0x240e0000  :  0xffffffff  0xffffffff  0xffffffff  0xffffffff

Block  8
Address 0x24100000  :  0xffffffff  0xffffffff  0xffffffff  0xffffffff

Block  9
Address 0x24120000  :  0xffffffff  0xffffffff  0xffffffff  0xffffffff

Block  10
Address 0x24140000  :  0xffffffff  0xffffffff  0xffffffff  0xffffffff
```


Example 8-5 DiagnosticList section command

```
AFU>DiagnosticList Section B1
 Block 1 Image Number 1 type 1 Used by image hello_world
 Block 2 Image Number 2 type 1 Used by image dhrystone
 Block 3 Image Number 2 type 1 Used by image dhrystone
Block Number 4 unused
Block Number 5 unused
Block Number 6 unused
Block Number 7 unused
Block Number 8 unused
Block Number 9 unused
Block Number 10 unused
```

8.3.4 TestBlock

This command tests the integrity of the block under test by writing a continually varying stream of words to the block and then reading the block out, and comparing the reads with the original data, displaying either `pass` or `fail` for the block.

The `TestBlock` command initially checks for data in the block conforming to the flash library specification, and does not allow any testing over a valid image. Example 8-6 shows the response to the command.

The block number must be included in the command line.

Syntax

`testblock b`*n*

where:

n is the logical block number to be tested.

Example 8-6

```
AFU> TestBlock B200
Do you really want to do this (Y/N)? y
deleting block 200
Writing test pattern to block 200
Reading test pattern from block 200
Flash test of block 200 worked
```

8.3.5 Delete

The `Delete` command deleted the full image from flash memory as selected. Once deleted, it cannot be retrieved. There is a final check to ensure the action is required. Example 8-7 shows the response to the command. You must input a valid image number or no action will be taken.

Syntax

`delete` *n*

where:

n Is the image number of the full image to be deleted.

Example 8-7

```
AFU> Delete 4
Do you really want to do this (Y/N)? y
Scanning Flash blocks for usage

Deleting flash image 4
Scanning Flash blocks for usage
```

8.3.6 DeleteBlock

This command deletes the specified block input on the command line, irrespective of any FLS images in flash. Example 8-8 shows the response to the command.There is a final user check to ensure the action was intended.

Caution

If used incorrectly, this command might damage images that span multiple blocks.

Syntax

`delete`block B*n*

where:

n Is the number of the specific block to be deleted.

Example 8-8

```
AFU> DeleteBlock B17
Do you really want to do this (Y/N)? y
Delete flash block 17
Scanning Flash blocks for usage
```

8.3.7 DeleteAll

This command erases all flash blocks, excluding block 255. This exclusion ensures that any SIBs being used remain intact. Example 8-9 shows the response to the command.

The `DeleteAll` action takes two minutes to complete on an ARM Integrator/CM920T board.

Syntax

`deleteall`

Example 8-9

```
AFU> deletea
Do you really want to do this (Y/N)? y
Deleting flash blocks:
This takes approximately 2 minutes

AFU>
```

8.3.8 Program

This command takes an image from a host computer and places it in the flash memory location that conforms with the flash library specification (see *Image management* on page 7-8). A footer and image information block appended to the image and header.

The AFU analyzes the input file, and tries to ascertain the storage address and image type from the file. If the image type is unrecognized, the AFU defaults to binary storage and store the image either directly in the location defined in the command line, or in the lowest available space within the flash blocks.

If the start location is omitted from the command input, the AFU uses the address taken from the header, or, if this is not available, the AFU will search for the lowest space large enough to house the image. The AFU always shows where the image is being stored (in block numbers). If the image executes from RAM, it can be placed anywhere in flash and the boot switcher will move it to RAM when it is run.

If the AFU discovers that the storage block is unavailable, it displays a warning and return. The AFU will not destroy any data found at the required address.

There is no restriction to the programming address of the image. The image can be programmed to start anywhere within a block.

The AFU will check for the image number input already in use, and will not allow the programming to take place if there is a duplication.

If there is an error in the command line, the complete command must be retyped.

Syntax

`program` *`n name path\filename`* `[location] [noboot]`

where:

`n`
: Is the unique number of the image to be programmed. This is a logical number and in not related to the order of the images in flash.

`name`
: Is the name, up to 16 characters, to identify the image being programmed. It does not need to be unique.

`path\filename`
: Is the path to the required file being programmed into the flash. The path and filename are retrieved using semihosting, so they must be the correct format for the host system.

`[`*`location`*`]`
: Is the optional address, or block number, of the start of the image in flash memory, using one of the following formats:

  - decimal base address
  - hexadecimal base address
  - block number specified as `B`*`n`*, where *`n`* is a logical flash block number.

`noboot`
: Is an optional flag to indicate to a boot switcher not to boot from this image.

Examples

In Example 8-10, a large image is programmed into a clean flash device. The unique image number is entered as `0`, and is named `Large_Image`. The image is retrieved from an MSDOS system (a backslash is used), and the file is named `large.bin`.

In this case, the image start address is omitted. The input file is ELF (`.axf`), so the AFU reads the start address from the image. The AFU shows that it has searched for the space, and gives the address and block number for storage.

As the image spans the blocks, the progress is shown. Finally, the flash device is scanned to update the image list, the new image is seen with the `List` command.

Example 8-10 Program Image command (large file)

```
AFU> Program 0 Large_Image d:\large.axf

Lowest available flash at location 0x24000000 block B0
The image load address is 0x24000000
Programming Block B0
Programming Block B1
Programming Block B2
Programming Block B3
Programming Block B4
Programming Block B5
Programming Block B6
Programming Block B7
Programming Block B8
Programming Block B9
Programming Block B10
Programming Block B11
Programming Block B12
Programming Block B13
Programming Block B14
Programming Block B15
Scanning Flash blocks for usage


AFU> list
Listing images in Flash
Image 0 Block 0 End Block 16 address 0x24000000 exec 0x24000000
- name Large_Image
AFU>
```

In Example 8-11, a small file is programmed into Block 18. The AFU does not search for available space because the location is specified in the command line. The image is a binary file so there is no alternative storage address in the header.

Example 8-11 Program Image command (small file)

```
AFU> Program 2 small_file d:\small.bin B18
Programming Block B18
Scanning Flash blocks for usage

AFU>
```

In Example 8-12, the display shows there is a conflict between an existing image and the execute address of a new image. The list shows that the lowest available block is Block 19, and that the user has tried to program the `hello.axf` file into this location. The image is then compiled for a different location, and the file header conveys this information.

The AFU prepares to insert the file at the correct execution address, but discovers that the block is being used (the block is part of Image 0). The AFU does not proceed with the download, but shows the error with the first block number involved. In this case, you must investigate the clash and decide what to do (in this case, you must either delete Image 0 or recompile `hello.axf`).

Example 8-12 List the outcome of Program Image

```
AFU> List

Listing images in Flash

Image 0 Block 0 End Block 16 address 0x24000000 exec 0x24000000 - name Large_Image
Image 1 Block 17 End Block 17 address 0x24220000 exec 0x24220000 - name hello
Image 2 Block 18 End Block 18 address 0x24240000 exec 0x24240000 - name small_file
AFU> Program 3 AIF_Image d:\hello.axf B19
The image Load address is 0x24020000 from the header
There is not enough space for the image found at this location
 As the image requires 0x00002f3c bytes
Please delete Block B1
AFU>
```


8.3.9 Read

This command takes an image from memory and stores it, in the original format, on the host computer. The original header is stored first, followed by the code body. The image is stored directly into *filename* on the host. The AFU does not alter the filename to reflect the image type or add any extension.

The AFU halts the file storage if there are any problems detected by the host.

Syntax

`read` *n filename*

where:

n Is the unique number of the image to be stored on the host computer.

filename Is the filename, with complete path, to the required file being written to the host computer. You must ensure that the path and filename are correct for the host system since they are stored using semihosting.

In Example 8-13, the image `Hello` is saved to the host system as `test.tst`. The file is the exact copy of the original programmed file, inclusive of headers.

Example 8-13

```
AFU> r 1 d:\test.tst
Reading Block Number  17 of image hello

AFU>
```

8.3.10 Quit

This command quits the current AFU session. After you quit the session, you must restart the program.

Syntax

`quit`

8.3.11 Help

This command displays the AFU command summary.

Syntax

help

In addition to h, you can type ? to display the command summary.

Example 8-14 shows the output from the Help command.

Example 8-14

```
AFU> Help
AFU command summary:

List                - List images in flash
DiagnosticList <All> | <Section Bn> | <Footer Bn> | <Dump Bn>  Bn = B<Block No.>
   - Lists information stored in the Flash by block, footer or block start dump
TestBlock B<block-number>
   - Write a test pattern to a particular flash block except block 255 (SIB Block)
Delete <image-number>
   - Delete an image in flash
DeleteBlock B<block-number>
   - Deletes a block that appears not to be in an image
DeleteAll
   - Deletes all blocks except block 255 (SIB Block)
Program <image-number> <image-name> <file-name> [<address> |or| B<block_no>]
[noboot]
   - Program the given image into flash at address, 0x<hex_addr> or block number
Read <image-number> <file-name>
   - Read the given image from flash into a file
Quit
   - Quit
Help
   - Print this help text
Identify
   - Identify Flash Type
Swap Device
   - Change active flash device
AFU>
```

8.3.12 Identify

This command identifies the current active flash device. It displays the known information (as shown at startup) for the currently selected (active) flash device.

Syntax

identify

Example 8-15 shows the output from the Identify command.

Example 8-15

```
AFU> Identify
Current Active Flash device is :-
INTEL   Flash device at 0x24000000 address : size 0x2000000
AFU>
```

8.3.13 Swap

This command allows you to change between the different FLS flash devices on the system.

The AFU cannot operate with non-FLS storage devices such as a boot device. These might be listed, but they cannot be selected using the `Swap Device` command.

Syntax

swap

In Example 8-16 on page 8-20, the only two options available are a boot-type flash (that is not accessed by the AFU, and is therefore unavailable) and the INTEL type flash. If `1` is selected, an error message results, so `2` is the only valid option.

Example 8-16

```
AFU> Swap
Searching for flash devices
Current flash devices found
1 . Boot    type at address  0x00000000
2 . INTEL   type at address  0x24000000
Please select active device by number....
2
Selected Flash Device
INTEL   type at address  0x24000000
Scanning Flash blocks for usage

AFU>
```


8.4 The Boot Flash Utility

The *ARM Boot Flash Utility* (BootFU) allows you to modify the specific boot flash sector on the system.

Caution

The Boot Flash sector on the Integrator board contains important system setup data (the FPGA initialization data) as well as the boot monitor and switcher code.

Modification of the boot flash on the Integrator board always involves a complete boot flash chip erase prior to programming. If the flash is programmed with incorrect data it halts operation of the board. This is generally a catastrophic failure.

If a problem is found with the downloaded data, the BootFU options can halt programming prior to erasing the flash device. This gives you a chance to backup the flash information.

In addition to diagnostic functions, BootFU can:

- update the whole boot area from an Intel hex file containing boot monitor and FPGA data
- update only the boot monitor area
- update only the FPGA area.

8.4.1 File Types

BootFU accepts `.aifbin`, `.elf`, `.bin`, and `.mcs` files for the downloaded image, although the filename and extension is not important because the BootFU code checks the file type from the data records transferred.

8.4.2 Setup

BootFU must be loaded into the target system RAM to operate. This is usually done using an ARM debugger, for example the *ARM Debugger for Windows* (ADW):

1. Connect the debugger to the board requiring a boot update.
2. Use the **Load Image** command to load the `bootfu.axf` into RAM at address `0x8000`.
3. Ensure the console window is active. If it is not, select **Console** from the **View** menu.
4. Run the utility by pressing **F5** or selecting **Execute→Go**.

The Console window shows a header message similar to:

```
ARM Firmware Suite
Copyright (c) ARM Ltd 1999-2000. All rights reserved.

Boot Flash Utility
Program Version 1.0
Date: 29 Jan 2000
```

The utility checks the available flash on the system and show the message:

```
Searching for flash devices
Flash device 1 found at 0x20000000 (4 blocks of size 0x20000)
Flash device 2 found at 0x24000000 (256 blocks of size 0x20000)
Device 1 found as Boot device
Scanning Flash blocks for usage
```

BootFU programs boot flash. Any flash not designated as `Boot` cannot be selected.

BootFU is ready for input when the `BootFU>` prompt is displayed. This is the input line for any of the commands.

8.5 BootFU commands

You can enter the commands shown in Table 8-6 at the `BootFU>` prompt.

Table 8-6 Commands

Command	Short form	Description
Help on page 8-24	`h` or `?`	Display commands
List on page 8-25	`l`	Lists the images currently programmed into flash
DiagnosticList on page 8-25	`dia`	Lists the first four words of the selected block or the selected block footer information
Program on page 8-27	`p`	Programs the boot flash
Read on page 8-29	`r`	Upload an image to the host file system
Quit on page 8-29	`q`	Quit the Boot Flash Utility
Identify on page 8-30	`i`	Identifies the current active flash device
Clear on page 8-30	`c`	Deletes any backup images stored in the system flash
Swap on page 8-30	`s`	Reserved for future expansion

8.5.1 Help

You can see a summary of the commands by typing `help`, `h` or `?`.

Syntax

`help`

Example 8-17 Help example

```
BootFU> ?
AFU command summary:

List                - List images in flash
DiagnosticList <Footer Bn> | <Dump Bn>  Bn = B<Block No.>
                    - Lists information stored in the Flash by footer
                      or block start dump
Program [i<image-number>] [*<image-name>] <file-name> [b<block_no>] [!]
                    - Program the given image into flash block number -
                      ! means no boot backup
Read <image-number> <file-name>
                    - Read the given image from flash into a file
Quit                - Quit
Help                - Print this help text
Identify            - Identify Flash Type
ClearBackup         - Removes any Boot backup images from the main system flash
Swap Device         - Change active flash device displayed as unformatted data.
```


8.5.2 List

This command lists the images currently programmed into flash. If the image has a header, its information is displayed. If there is only unstructured data in the flash, it is displayed as unformatted data. Example 8-18 shows the response to the command.

Syntax

`list`

Example 8-18 List Example

```
BootFU> list
Block   0 Image Number   0 type 1 Used by image Boot_Monitor
Block   1 is unused
Block   2 Has unformatted data
Block   3 Has unformatted data
```

In Example 8-18, the boot monitor has footer information applied to it as Image1. The FPGA setup data in the upper two blocks never has footer information applied.

If the entire boot area is programmed from Intel hex files, there is no footer information. The listing only shows the programmed blocks as unformatted data.

8.5.3 DiagnosticList

The DiagnosticList command allows the listing of the first four words of the selected block or the selected block footer information. Example 8-19 shows the response to the command.

Syntax

`diagnosticList f b`*n*`|d b`*n*

where:

n is the unique number of the block.

`f` lists the block footer of the selected block.

`d` dumps the first four words of the selected block.

Example 8-19 DiagnosticList Example

```
BootFU> dia f b0
Footer for Block   0 at Address 0x20000000
infoBase  : 0xffffffff
blockBase : 0xffffffff
signature : 0xffffffff
type      : 0xffffffff
checksum  : 0xffffffff

Footer for Block   1 at Address 0x20020000
infoBase  : 0xffffffff
blockBase : 0xffffffff
signature : 0xffffffff
type      : 0xffffffff
checksum  : 0xffffffff

Footer for Block   2 at Address 0x20040000
infoBase  : 0xadffbfbf
blockBase : 0xff6fdff6
signature : 0x9ffeffdf
type      : 0xfcffefef
checksum  : 0xfffffffe

Footer for Block   3 at Address 0x20060000
infoBase  : 0xb5deebb5
blockBase : 0xebb55feb
signature : 0x5bebb55e
type      : 0xf8dafff5
checksum  : 0xff847a08
```

8.5.4 Program

This is the most important command in the BootFU as it starts programming the boot flash. It is also potentially the most damaging. The command requires at least the path and filename parameters.

All of the options are position-independent but it is recommended that the binary-only options are included for any binary downloaded files.

Syntax

`program` *`path filesep filename`* [`b`*`blnum`*] [`i`*`imnum`*] [`*`*`string`*] [`!`]

where:

`path` is the path to the file.

`filesep` is the file separator used on the host operating system.

`filename` is the name of the file.

`blnum` is the block number to be programmed.

`imnum` is the image number for the footer information (binary files only).

`string` is the name of the image for the footer information (binary files only).

`!` specifies not to backup the boot area.

Caution

The `!` option speeds up the program time but does not allow download problems or partial downloads to complete correctly. It should only be used if you are sure about the file being downloaded.

Examples

Example 8-20 on page 8-28 shows a complete boot area program from an Intel hex file.

Example 8-20 Program boot area

```
BootFU> program d:\test.mcs
************************************************************
* WARNING: re-programming the Boot Flash can cause the system   *
*          to cease operation - if the images are corrupted or  *
*          incorrect. Are you sure you wish to continue         *
************************************************************

Do you really want to do this (y/N)? y
Image will be stored with no footer information
Backing up boot image
Boot Image backed up to board flash
Deleting Boot Flash area
Decoding and Writing .mcs type file
Scanning Flash blocks for usage
BootFU>
```

In Example 8-21 the boot monitor code in block 0 is being updated from a binary file. The system FPGA data is restored from the backup image stored in the main system flash.

Example 8-21 Program block 0

```
BootFU> program b0 i0 *Boot_Monitor d:\boot.bin
************************************************************
* WARNING: re-programming the Boot Flash can cause the system   *
*          to cease operation - if the images are corrupted or  *
*          incorrect. Are you sure you wish to continue         *
************************************************************

Do you really want to do this (y/N)? y
Backing up boot image
Boot Image backed up to board flash
Deleting Boot Flash Area
Writing Binary type file
Programming Block B0
Restoring unprogrammed boot flash from Backup
Deleting Backup
Scanning Flash blocks for usage
BootFU>
```


BootFU operation includes checks to ensure that the correct data is used to update the image.

The standard operation of BootFU is usually either:

`program` *`path/filename`*`.mcs`

This is for the complete update of the boot flash.

`program` *`path/filename`* `b`*`nin*string`*

For updates, the identifier parameter is optional for image recognition of binary files.

The *no backup* option (`!`) is not recommended. It reduces the program time but it is not as safe as backing up the data in the system flash.

8.5.5 Read

This command allows an image (specifying the image number as `i`*`number`*) or a block (specifying the block number as `b`*`number`*) to be uploaded to the host file system. You must add the path and filename parameters to the command. If block 0 is requested the entire boot device is uploaded and saved. The output file is a pure binary file.

Syntax

`read i`*`n`*`| b`*`n`*

where:

`n` is the unique number of the image block.

`i` reads the selected image.

`b` reads the selected block.

8.5.6 Quit

This command quits the BootFU.

Syntax

`quit`

8.5.7 Identify

This command identifies the current active flash device. This displays the flash type (boot), device physical base address, and device size in bytes.

Syntax

```
identify
```

8.5.8 Clear

This command deletes any backup images stored in the system flash. The backup images are automatically cleared by the utility when the boot flash is fully programmed. Use this option if there has been a catastrophic (power) failure during programming and the backup file has not been removed. The clear command deletes all backup files programmed into the system flash.

Syntax

```
clear
```

8.5.9 Swap

This command is reserved for future expansion where the BootFU might be used on systems with partitioned boot flash or multiple boot flash devices. For the Integrator board this option is redundant as there is only one boot device to select.

8.5.10 BootFU Warning messages

If a binary file is downloaded with no block number, it is placed at block 0 by default. The warning message in Example 8-22 is displayed with the option to quit the program:

Example 8-22 Download warning message

```
*************************************************************
* WARNING: A binary file has been input without specifying the  *
*          target block, if you wish to proceed the block number*
*          will default to 0 - if not the boot sector flash will*
*          be restored from the backup                          *
*************************************************************
Do you really want to do this (y/N)?
```

If the downloaded file is a binary file and no backup has been requested, the warning message in Example 8-23 will be displayed with an option to quit the program:

Example 8-23 Binary warning message

```
*************************************************************
* WARNING: A binary file has been input without specifying the  *
*          target block, if you wish to proceed the block number*
*          will default to 0 - if not the boot sector flash will*
*          be restored from the backup                          *
*************************************************************
Do you really want to do this (y/N)?
```


Chapter 9
PCI Management Library

This chapter describes the *Peripheral Component Interconnect* (PCI) library and how you can use it to configure PCI subsystems. Its contains the following sections:

9.1 About PCI

This section provides an introduction to the PCI terminology used in this chapter. Figure 9-1 shows the major components of an example PCI system.

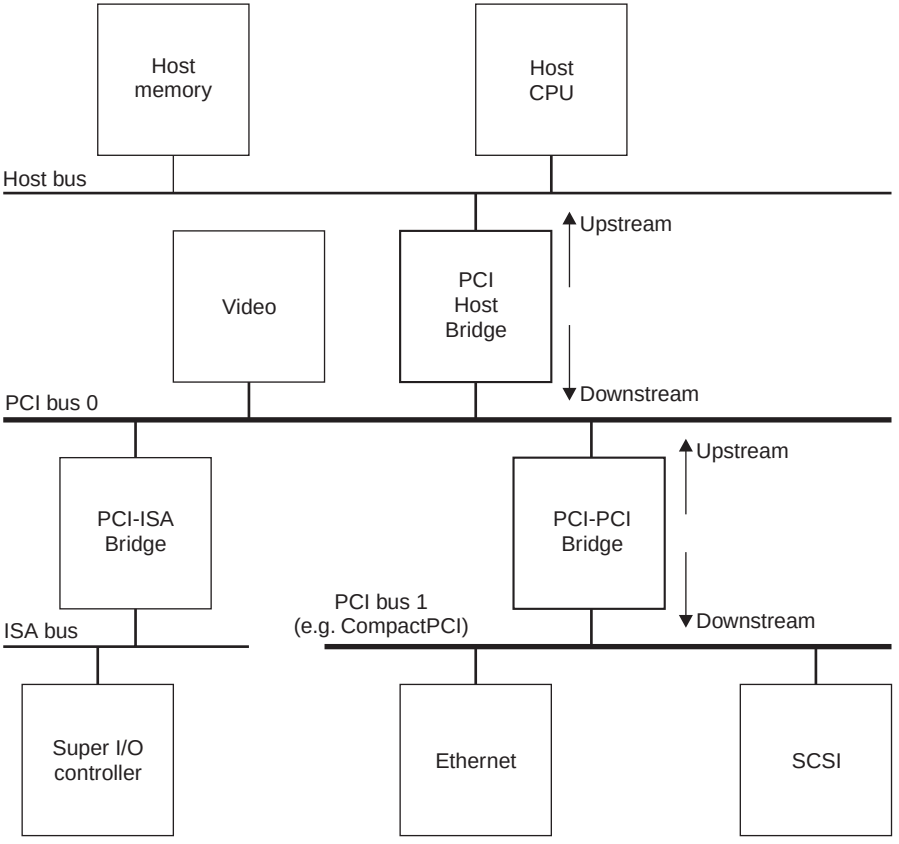

Figure 9-1 An example PCI system

The system features illustrated in Figure 9-1 are:

Host bus In this system, the CPU and host memory reside on the host bus.

Host bridge This is a device that allows transactions between the host bus and PCI bus to take place. These typically support a variety of reads and writes in both directions and might incorporate FIFOs to support writes in both directions. The types of transactions supported by the bridge are configurable.

In the case of the ARM Integrator, there is an additional bridge between the host bus and system bus to which the processors and memory connect. However, from the point of view of the PCI functions, this bridge is transparent.

PCI-PCI bridge The electrical loading on a PCI bus is limited and there is a limited number of devices that can be connected. To overcome this, multiple PCI buses can be used. The different buses are connected through PCI-PCI bridges. In this system, the PCI-PCI bridge connects between bus 0 (used to access fast on-board peripherals) and bus 1 (in this case is a CompactPCI backplane bus).

All devices connected to the PCI buses including bridges are uniquely identified by the number of the bus to which they are attached and the slot number they occupy on that bus. Typically, the CPU or host bridge is in slot 0.

In the case of a multi-function PCI device, such as a combined sound and video device, each function is treated as a different device. In order to uniquely address a PCI device, specify the bus, slot, and function numbers for that device.

PCI-ISA bridge The PCI-ISA bridge provides support for legacy devices. In this example, a super input/output controller is used. The PCI-ISA bridge translates PCI address cycles into ISA address cycles so that the CPU can access the legacy devices on the ISA bus.

Primary bus In this system, PCI bus 0 is the *primary* (or *upstream)* bus for the PCI-PCI bus. The primary bus for a particular bridge is the bus nearer to the host CPU that controls the system.

Secondary bus In this system, PCI bus 1 is the *secondary* (or *downstream*) bus for the PCI-PCI bridge.

The bus numbering is important. During initialization the bus numbers are assigned by the CPU. However, device drivers do not differentiate when communicating with devices on different PCI buses.

9.2 PCI configuration

This section provides a brief software-biased overview of PCI configuration. The PCI library contains software to fully configure PCI subsystems. This includes:

- scanning and identifying PCI devices on local and bridged PCI buses
- assigning device resources in PCI memory and I/O space
- allocating interrupt numbers
- numbering the PCI-PCI bridges.

The PCI library uses services exported from the µHAL library to access the PCI subsystem in a generic way (see *About µHAL PCI extensions* on page 9-15).

The PCI component of the firmware base level contains the PCI library and example applications. See the sources for the `scanpci` application that initializes the PCI subsystem and displays its topology. A sample device driver is also provided that initializes the PCI bus and assigns interrupt handlers (see *Example PCI device driver* on page 9-23).

9.2.1 PCI address spaces

There are three PCI address spaces:

- configuration space
- I/O space
- memory space.

Configuration space

Each PCI device in the system has a 256-byte header in PCI configuration space. The contents of this header are specified by the PCI standard and defines, among other things:

- the device type
- the device manufacturer
- how much PCI I/O space the device requires
- how much PCI memory address space the device requires.

The address of a PCI Configuration header for a device is directly related to the location of the device in the PCI topology. The system initialization code must locate the PCI devices in the system by looking at all of the possible PCI configuration headers in PCI Configuration space. The PCI configuration code is run by the host bridge. That is, the processor that owns PCI bus 0.

To find the slot a PCI device is in, the CPU reads the first 32 bits of the PCI header for the device by issuing a Type 0 PCI Configuration Cycle, (see Figure 9-2 on page 9-5). Each slot is addressed by setting one of bits [31:11]. For example, slot 0 is found by issuing a Type 0 PCI Configuration Cycle with bit 11 set high.

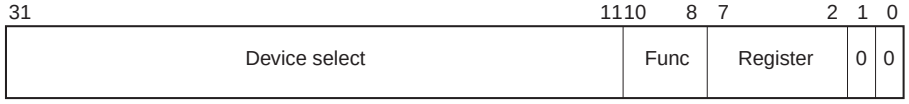

Figure 9-2 PCI Type 0 configuration cycle

The format of a PCI configuration header for a device is shown in Figure 9-3.

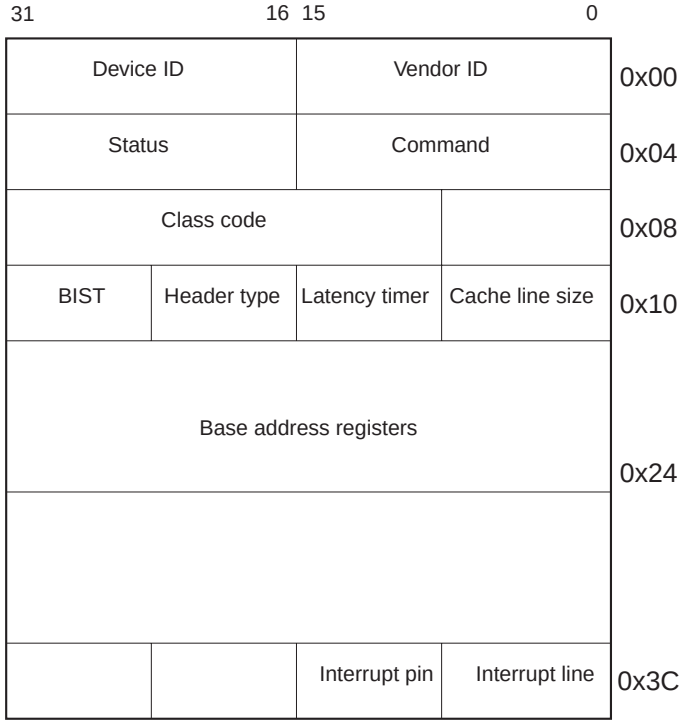

Figure 9-3 PCI configuration header

The device and vendor identifiers are unique and completely identify the maker of the PCI device and its type. In addition, the class code identifies the generic type of the device (for example, video device). The Base Address Registers are used to request and grant space in PCI I/O or memory spaces.

I/O space

PCI I/O space is used for small amounts of memory that the device makes accessible to the main processor. Typically, this contains registers within the device.

Memory space

PCI memory space is used for much larger amounts of memory. Video devices often use large amounts of PCI memory space.

9.2.2 PCI-PCI bridges

The PCI initialization code must recognize PCI-PCI bridges and configure them so that that addresses and data are passed between the upstream and downstream sides.

Except for the required initialization code, a PCI-PCI bridge is transparent to the PCI devices in the system. When a PCI device is granted an address range in PCI I/O or PCI Memory space, software running on the host bridge does not know the PCI bus that the device is connected to. PCI I/O and PCI Memory address spaces do not have a hierarchy.

The PCI configuration code uses a Type 1 PCI Configuration Cycle for addressing PCI devices that are not on the primary bus (see Figure 9-4).

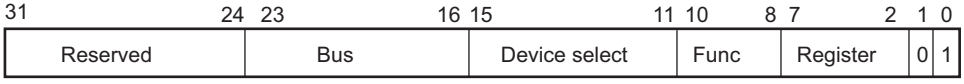

Figure 9-4 PCI Type 1 configuration cycle

The Type 1 Configuration cycle includes the bus number within the address.

The PCI-PCI bridges (between the host bridge and the final PCI bus to which the target device is attached) are responsible for passing the Type 1 cycle along to the next bus. The algorithm for this mechanism is:

- If the configuration cycle is for a device on the downstream bus, translate it to a Type 0 cycle.
- If the Configuration cycle is for a device beyond the downstream bus, pass it on to the next bridge unchanged (as a Type 1 cycle).

This means that the buses must be numbered in a particular order. When the type 1 PCI configuration cycle reaches its destination bus, the final PCI-PCI bridge translates it into a Type 0 configuration cycle.

9.3 The PCI library

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.

9.3.1 Initializing the PCI subsystem

This is carried out in three phases:

1. Perform any host bridge initialization (using the system specific µHAL support function).
2. Scan the local PCI bus for PCI devices. Some of the PCI devices found are PCI-PCI bridges and, in this case, the PCI initialization code also scans for PCI devices downstream of the PCI-PCI bridge. In doing this the code must number the PCI buses.
3. Assign resources to the PCI devices. These resources are:
   - areas of PCI I/O and PCI memory. PCI devices must be granted addresses in PCI I/O and PCI Memory space and those addresses must be enabled.
   - interrupt numbers. PCI devices must be given relevant interrupt numbers that are meaningful to the device drivers in the application or operating system.

9.3.2 Data structures

As the initialization code locates PCI devices, it builds a `PCIDevice` data structure describing each one. These each have the format shown in Example 9-1.

Example 9-1 Building data structures

```
/* A PCI device, the PCI configuration code builds a list of PCI devices */
typedef struct PCIDevice {
    struct PCIDevice *next ;    // next PCI device in the system (all buses)
    struct PCIDevice *sibling ; // next device on this bus
    struct PCIDevice *parent ; // this device's parent device Flags
struct {
unsigned int bridge : 1 ;      // This is a PCI-PCI bridge device
unsigned int spare : 15 ;
} flags ;
// This part of the structure is only relevant if this is a PCI-PCI bridge
struct {
struct PCIDevice *children ;    // pointer to child devices of this PCI-PCI bridge
    unsigned char number ;      // This bus's number
    unsigned char primary ;     // number of primary bridge
    unsigned char secondary ;   // number of secondary bridge
    unsigned char subordinate ; // number of subordinate buses
} bridge ;
// Vendor/Device is a unique key across all PCI devices.
unsigned short vendor ;
unsigned short device ;
// PCI Configuration space addressing information for this device
unsigned char bus ;
unsigned char slot ;
unsigned char func ;
} PCIDevice_t ;
```

The list is hierarchical, reflecting the PCI topology of the system. If the PCI device is a PCI bridge, its `children` pointer points at the first PCI device found downstream of it. Each PCI device is on two lists:

`PCIroot` Points at the host bridge

`PCIDeviceList` Points at all of the PCI devices in the system.

To find all of the PCI devices in the system, follow the address in `PCIDeviceList` through the next pointer of each `PCIDevice` structure.

Figure 9-5 on page 9-9 shows the `PCIDevice` structures for part of the example PCI system. `PCIRoot` points at a host bridge that has two children (a PCI-ISA bridge and a PCI Video device). `PCIDeviceList` points first at the host bridge and then at each of the PCI devices in the system. For simplicity, the parent pointer for the PCI-ISA bridge and video device is omitted from the figure.

The storage space for these data structures is either statically allocated or, if the system supports it, allocated from µHAL heap storage.

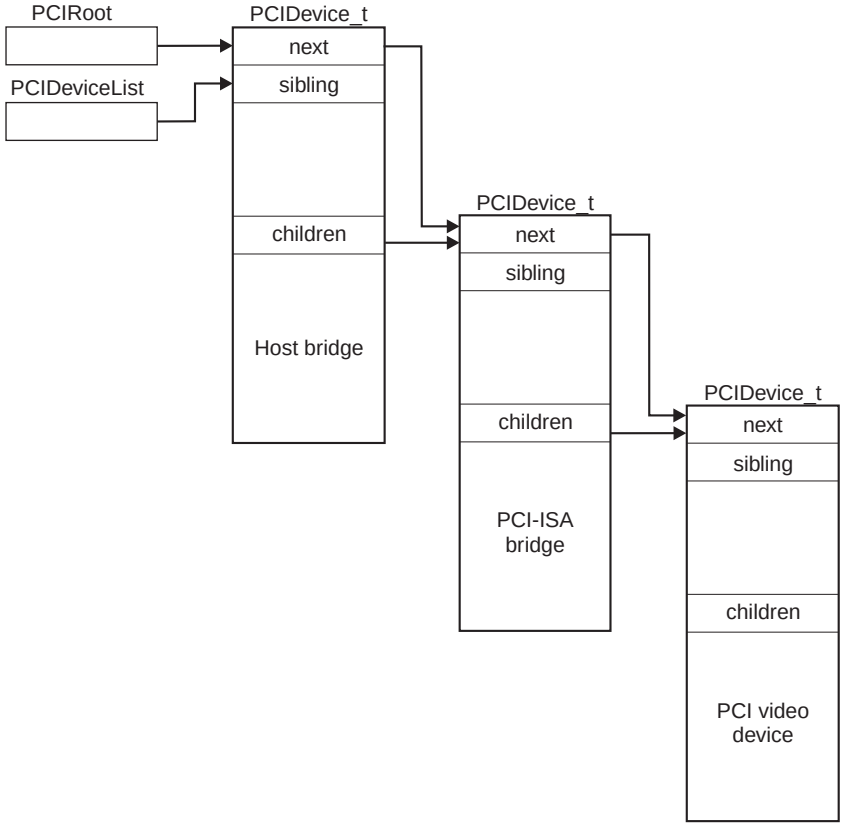

Figure 9-5 PCI library data structure

Note

These data structures are not exported beyond the PCI library, they are for internal use and must not be used outside the library.

9.3.3 Host bridge initialization

This function is board-specific and contained in the function `uHALir_PCIInit()`. This function initializes the PCI host and enables the PCI access functions and primitives to function. This function is expected to be able to safely re-initialize the PCI subsystem.

9.3.4 Scanning the PCI system

During scanning, the PCI initialization code:

1. Builds a `PCIDevice` data structure describing the host bridge.
2. Issues Type 0 configuration cycles looking for all of the devices attached to this bus.
3. Builds a `PCIDevice` data structure for each device it finds and adds it as a child of bus 0 and into `PCIDeviceList`.

If the device is a multi-function device (as indicated by the Header Type field of the PCI Configuration Header), the scanning code creates one `PCIDevice` data structure for each function.

If the device is a PCI-PCI bridge, the scanning code scans the downstream PCI buses looking for more PCI devices and bridges. This depth-wise recursive algorithm is used in order that the buses attached each PCI-PCI bridge can be correctly numbered.

The scanning phase is complete when:

- the PCI library has a built tree of `PCIDevice` data structures that describe the topology of the PCI subsystem
- the PCI buses have been numbered.

9.3.5 Assigning resources to PCI devices

The next phase is to assign areas of PCI I/O and PCI Memory and, if necessary, an interrupt number to each of the PCI devices in the system.

PCI-PCI bridges must be configured to allow downstream accesses of PCI I/O and PCI Memory for those devices attached to their secondary PCI bus.

Assigning PCI I/O and Memory areas

The PCI Configuration header for each device contains a number of *Base Address Registers* (BARs). These describe the type of PCI address space the device requires and how much it requires. The device initialization code requests this information by writing 1s to all bits of each BAR in the device and reading back the result.

If the device returns a nonzero value, the PCI initialization code must assign it an area of PCI I/O or PCI Memory space according to the value of bit 0. If bit 0 is:

`0` the request is for PCI I/O space.

`1` the request is for PCI memory space.

The PCI library assigns the next area of the address space that the device has requested and enables access to that type of memory (using the Command field of the PCI Configuration Header).

The location of a device is defined by writing the assigned address back to the appropriate BAR. Figure 9-6 shows the format of PCI Memory space addresses.

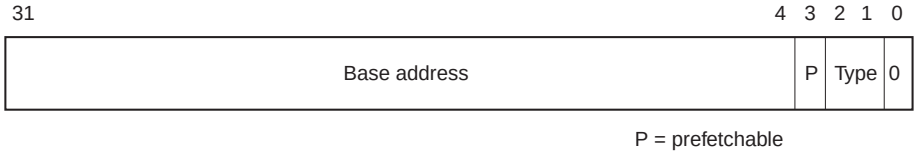

Figure 9-6 Base address for PCI Memory space

Figure 9-7 shows the format of PCI I/O space address.

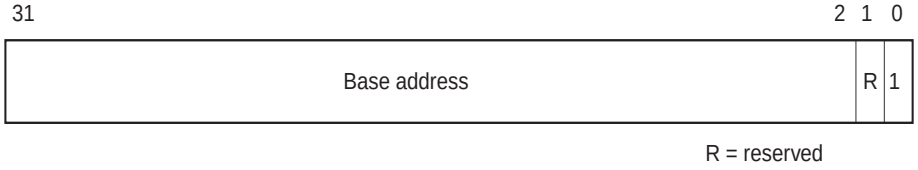

Figure 9-7 Base address for PCI I/O space

The PCI configuration code ensures that assigned addresses are naturally aligned. For example, if a PCI device requests `0x40000` bytes of PCI Memory, they align the allocated address on a `0x40000` byte boundary.

PCI-PCI bridges pass PCI Memory and PCI I/O cycles from their upstream side to the downstream side if the address is assigned to one of the downstream devices. Each PCI-PCI bridge stores two pairs of addresses in its BARs that define the upper and lower bounds of the PCI Memory and PCI I/O address spaces downstream of the bridge.

Access to these two spaces must be enabled by using the command field of the PCI Configuration header. All PCI I/O and PCI memory addresses downstream, including those beyond any downstream PCI-PCI bridges, must be assigned within these spaces. Address assignment is carried out using a recursive algorithm with addresses beyond the farthest bridges being assigned first.

Assigning interrupt numbers

The PCI specification describes the function of the interrupt line field of the PCI configuration header as operating system dependent, but intended to pass interrupt routing information between the operating system and the device driver. During PCI initialization, the PCI set-up code writes a value into the field. Later, when the device

driver initializes the device, it reads the value and passes it to the operating system, requesting an interrupt when the triggering event occurs. When an interrupt is triggered, the operating system must route the interrupt to the correct device driver.

Typically, this value is an offset into the *Programmable Interrupt Controller* (PIC). For example, the value 5 would mean bit 5 of the PIC. It is not important what the number is, but the operating system interrupt handling code and the PCI setup code must agree on the meaning.

Each PCI device has four interrupt pins labelled A, B, C, and D. The interrupt pin used by a device is defined in the PCI configuration header for the device in the interrupt pin field. The routing of interrupts between slots and the interrupt controller is entirely system specific. For this reason, the PCI library uses a board-specific function, `uHALir_PCIMapInterrupt()`, to find out how interrupts are routed on a particular board.

The interrupt controller might have as few as four PCI interrupts (one per pin) or as many as (number of slots x 4). PCI interrupts might be shared by other devices. In other words, they must allow for their interrupt handler being called with no work to do.

Interrupts from downstream devices are routed through each bridge. Depending on the slot number of the device, the interrupt pin might be transposed as it crosses the bridge. For example, a PCI device interrupting on downstream pin B basic can cause pin C on the upstream side of the bridge to be asserted. The algorithm for working out the upstream interrupt pin that is asserted given a downstream slot number and interrupt pin is:

```
upstream_pin = (((downstream_pin - 1) + slot) % 4) + 1
```

where Pin A is 1, B is 2, C is 3, and D is 4. A value of 0, means default (Pin A).

The PCI initialization code applies this algorithm once for each PCI-PCI bridge between the PCI device and the host bridge. When it reaches bus 0, it takes the final pin number and the slot number of the PCI-PCI bridge and calls `uHALir_PCIMapInterrupt()` to return the interrupt number for the device.

9.3.6 Rebuilding the PCI library

Use the makefile in the PCI subdirectory of the source directory for your board to rebuild the library. For example, use `unix\source\Integrator940T\Pci\makefile` to rebuild the library for the Integrator board using the 940T processor.

For general information on makefiles and directory structure, see *Building the µHAL library* on page 2-8.

9.4 PCI library functions and definitions

The PCI library provides three functions and a number of definitions. These are all contained in `PCI/Sources/pcilib.h`. The PCI library is also used within the boot monitor component. Currently the PCI Library functions on the EBSA-285 (StrongARM-based evaluation board) and on ARM Integrator systems.

The functions are described in

- *PCIr_Init(void)*
- *PCIr_ForEveryDevice*
- *PCIr_FindDevice*
- *PCI definitions* on page 9-14.

9.4.1 PCIr_Init(void)

This function initializes the PCI subsystem by calling the system-specific `uHALir_PciInit()` function.

Syntax

```
void PCIr_Init(void)
```

9.4.2 PCIr_ForEveryDevice

This function calls the given function once for every PCI device in the system passing the bus, slot, and function numbers for the device. No ordering of devices should be assumed

Syntax

```
void PICir_ForEveryDevice (void (action) ( unsigned int,
                    unsigned int,  unsigned int))
```

9.4.3 PCIr_FindDevice

This function finds a particular instance of the PCI device given its vendor and device identifier.

Syntax

```
int PCIr_FindDevice(unsigned short vendor,
                    unsigned short device,
                    unsigned int instance, unsigned int *bus,
                    unsigned int *slot, unsigned int *func)
```

where:

vendor is the vendor identifier.

instance is the instance number.

device is the device identifier.

bus is the PCI bus to which the device is attached.

slot is the slot number of the device.

func is the function of the device.

Return value

0 If it has found the device. The bus, slot, and function number for the device is set up.

nonzero There is not an Nth instance of such a device.

9.4.4 PCI definitions

There are a number of system-specific PCI definitions that are used by the PCI library. These are listed in Table 9-1.

Table 9-1 PCI definitions

Definition	Function
UHAL_PCI_IO	The local bus address that PCI I/O space has been mapped to
UHAL_PCI_MEM	The local bus address that PCI Memory space has been mapped to
UHAL_PCI_ALLOC_IO_BASE	The address in PCI I/O space that the PCI address allocation should start allocating from
UHAL_PCI_ALLOC_MEM_BASE	The address in PCI Memory space that the PCI address allocation should start allocating from
UHAL_PCI_MAX_SLOT	The maximum number of PCI slots available on PCI bus 0

9.5 About µHAL PCI extensions

The ARM Firmware PCI library is independent of the particular system that it is running on. This means that it relies on board-specific code within the µHAL library to initialize the PCI subsystem.

The µHAL PCI extensions provide the following functionality to the PCI library:

Host bridge initialization

This system-dependent initialization is usually performed at system startup and involves setting up the host bridge interface (for example the V360EPC chip on the Integrator system) so that the generic PCI library can access all three areas of PCI memory. This code is held in `uHAL\Boards\<board>\board.c` and `uHAL\Boards\<board>\driver.s`.

Access primitives Access primitives allow access to the PCI memory spaces. These are functions and C macros that allow code to access areas of PCI memory without knowing how these areas of memory are mapped to and from local bus memory.

Each PCI supporting target must supply a set of functions that allow access to the three PCI address spaces. Within these functions the target software might need to perform system-specific operations. This system-specific code is external to the PCI library. The set of functions that are supplied as board-specific code (in `uHAL\Boards\<board-name>\board.c`) are described later in this section.

Interrupt routing Each PCI supporting board must supply a function that returns the interrupt number that is associated with the given PCI slot and interrupt pin. This information is used by the PCI library as it assigns resources to individual PCI devices.

PCI resource allocation

The µHAL library for a PCI supporting board exports code and definitions in the µHAL definition file `uHAL\h\uhal.h`.

9.6 µHAL PCI function descriptions

The standard µHAL library for a particular system includes system-specific PCI extensions to µHAL and system-specific initialization code. This section describes the following µHAL PCI functions:

- *uHALir_PCIInit*
- *uHALr_PCIHost* on page 9-16
- *uHALr_PCICfgRead8* on page 9-17
- *uHALr_PCICfgRead16* on page 9-17
- *uHALr_PCICfgRead32* on page 9-18
- *uHALr_PCICfgWrite8* on page 9-18
- *uHALr_PCICfgWrite16* on page 9-19
- *uHALr_PCICfgWrite32* on page 9-19
- *uHALr_PCIIORead8* on page 9-19
- *uHALr_PCIIORead16* on page 9-20
- *uHALr_PCIIORead32* on page 9-20
- *uHALr_PCIIOWrite8* on page 9-20
- *uHALr_PCIIOWrite16* on page 9-21
- *uHALr_PCIIOWrite32* on page 9-21
- *uHALir_PCIMapInterrupt* on page 9-21.

9.6.1 uHALir_PCIInit

This function initializes the host bridge, for example the V360EPC chip on the Integrator. This board-specific code is not normally called by an application (therefore it has a `uHALir` prefix). Rather, it is called by the PCI library initialization code `PCIr_Init()`.

Syntax

```
void uHALir_PCIInit(void)
```

9.6.2 uHALr_PCIHost

This function tests the board for PCI support.

Syntax

```
unsigned  char uHALr_PCIHost(void)
```


Returns

TRUE If the system supports PCI.

FALSE Otherwise.

9.6.3 uHALr_PCICfgRead8

This function reads 8 bits from PCI Configuration space.

Syntax

```
volatile unsigned char uHALr_PCICfgRead8(
                    unsigned int bus, unsigned int slot,
                    unsigned int func, unsigned int offset)
```

where:

bus is the PCI bus to which the device is attached.

slot is the slot number of the device.

func is the function of the device.

offset is the register offset of the device.

Returns

The 8-bit char from the configuration space.

9.6.4 uHALr_PCICfgRead16

This function reads 16 bits from PCI Configuration space.

Syntax

```
volatile unsigned short uHALr_PCICfgRead16(
                    unsigned int bus, unsigned int slot,
                    unsigned int func, unsigned int offset)
```

where:

bus is the PCI bus to which the device is attached.

slot is the slot number of the device.

func is the function of the device.

offset is the register offset of the device.

Returns

The 16-bit short from the configuration space.

9.6.5 uHALr_PCICfgRead32

This function reads 32 bits from PCI Configuration space.

Syntax

```
volatile unsigned int uHALr_PCICfgRead32(unsigned int bus,
                    unsigned int slot, unsigned int func,
                    unsigned int offset)
```

where:

bus	is the PCI bus to which the device is attached.
slot	is the slot number of the device.
func	is the function of the device.
offset	is the register offset of the device.

Returns

The 32-bit word from the configuration space.

9.6.6 uHALr_PCICfgWrite8

This function writes 8 bits to PCI Configuration space.

Syntax

```
void uHALr_PCICfgWrite8(unsigned int bus, unsigned int slot,
                    unsigned int func, unsigned int offset,
                    unsigned char data)
```

where:

bus	is the PCI bus to which the device is attached.
slot	is the slot number of the device.
func	is the function of the device.
offset	is the register offset of the device.
data	is the data written to the device.

9.6.7 uHALr_PCICfgWrite16

This function writes 16 bits to PCI Configuration space.

Syntax

```
void uHALr_PCICfgWrite16(unsigned intbus,
                         unsigned int slot,  unsigned intfunc,
                         unsigned intoffset, unsigned shortdata)
```

where:

bus	is the PCI bus to which the device is attached.
slot	is the slot number of the device.
func	is the function of the device.
offset	is the register offset of the device.
data	is the data written to the device.

9.6.8 uHALr_PCICfgWrite32

This function writes 32 bits to PCI Configuration space.

Syntax

```
void uHALr_PCICfgWrite(unsigned int bus, unsigned int slot,
                       unsigned int func,  unsigned int offset,
                       unsigned int data)
```

where:

bus	is the PCI bus to which the device is attached.
slot	is the slot number of the device.
func	is the function of the device.
offset	is the register offset of the device.
data	is the data written to the device.

9.6.9 uHALr_PCIIORead8

This function reads 8 bits from PCI I/O space.

Syntax

```
volatile unsigned char uHALr_PCIIORead8(unsigned int offset)
```

where:
offset is the address.

Returns

The 8-bit char from the I/O space.

9.6.10 uHALr_PCIIORead16

This function writes 16 bits from PCI I/O space.

Syntax

```
volatile unsigned short uHALr_PCIIORead16(unsigned int offset)
```

where:
offset is the address.

Returns

The 16-bit short from the I/O space.

9.6.11 uHALr_PCIIORead32

This function reads 32 bits from PCI I/O space.

Syntax

```
volatile unsigned int uHALr_PCIIORead32(unsigned int offset)
```

where:
offset is the address.

Returns

The 32-bit int from the I/O space.

9.6.12 uHALr_PCIIOWrite8

This function writes 8 bits to PCI I/O space.

Syntax-

```
void uHALr_PCIIOWrite8(unsigned int offset, unsigned char data)
```


where:

offset is the register offset of the device.

data is the data written to the device.

9.6.13 uHALr_PCIIOWrite16

This function writes 16 bits to PCI I/O space. The address is given by the *offset* argument.

Syntax

```
void uHALr_PCIIOWrite16(unsigned int offset,
                        unsigned short data)
```

where:

offset is the register offset of the device.

data is the data written to the device.

9.6.14 uHALr_PCIIOWrite32

This function writes 32 bits to PCI I/O space. The address is given by the *offset* argument.

Syntax

```
void uHALr_PCIIOWrite32(unsigned int offset, unsigned int data)
```

where:

offset is the register offset of the device.

data is the data written to the device.

9.6.15 uHALir_PCIMapInterrupt

This function returns the interrupt number associated with this PCI slot and interrupt pin.

Syntax

```
unsigned char uHALir_PCIMapInterrupt(unsigned char pin,
                                     unsigned char slot)
```

where:

pin is the bit position of the interrupt in the programmable interrupt controller for the system.

slot is the slot number of the device.

Returns

The interrupt number as an 8-bit `char`.

9.7 Example PCI device driver

The PCI component of the ARM Firmware Suite contains an example PCI device driver (in `.../Sources/example-driver.c`). This demonstrates how a device driver:

- finds the device
- examines its registers
- takes control of its interrupt.

These steps are carried out as follows:

1. Check that the system supports PCI (or is a PCI host):

```
/* Must be PCI host to initialise the bus */
  if (!uHALr_PCIHost ()) {
    uHALr_printf ("Not PCI host - can't scan the bus \n");
    return (OK);
}
```

2. If the system is a PCI host, initialize the PCI subsystem:

```
/* initialise the bus */
  uHALr_printf ("Initialising PCI");
  PCIr_Init ();
  uHALr_printf ("...done \n");
```

3. Scan the system for the PCI device of interest. In this example, a Digital 21142 ethernet device (with a vendor ID of `0x1011` and a device ID of `0x0019`):

```
/* look for the Digital 21142 ethernet device */
if (PCIr_FindDevice(DIGITAL, TULIP21142, 0, &bus, &slot,
                    &func) == 0) {
    unsigned int ioaddr, memaddr, irq ;
    int i ;
```

   The instance number in this case is 0 because the code is looking for the first instance. To find the next instance, make another call to `PCIr_FindDevice()` but with an instance of 1.

4. If the device is found, print out the location of its command and status registers (CSRs) in PCI I/O and PCI Memory. The code is shown in Example 9-2.

Example 9-2

```
/* found it, tell the world */
uHALr_printf("Found Digital 21142 ethernet device [%02d:%02d:%02d]\n",
       bus, slot, func) ;
/* work out the location of its CSRs in PCI IO and PCI Memory */
ioaddr = uHALr_PCICfgRead32 (bus, slot, func, PCI_MEM_BAR);
ioaddr &= ~0x0F ;
memaddr = uHALr_PCICfgRead32 (bus, slot, func, PCI_MEM_BAR+ 4);
memaddr &= ~0xF ;
uHALr_printf("\tCSRs are at 0x%08X (IO) and 0x%08X (Memory)\n",ioaddr,
        memaddr) ;
```

The addresses are from the PCI configuration header for the device. The device is addressed using the PCI bus number, slot number and function number returned by the call to `PCIr_FindDevice()` in the previous step.

5. Make calls to read the device CSRs from PCI I/O space. The CSRs are 64-bit aligned:

```
/* print out its CSRs (all 15) */
for (i = 0; i < 15; i++) {
   uHALr_printf("\t\tCSR %02d: %08X\n", i,
     uHALr_PCIIORead32(ioaddr + (i << 3))) ;
}
```

6. Find the interrupt number associated with this device from the PCI configuration header.

```
/* Find its interrupt number and assign it */
irq = uHALr_PCICfgRead8 (bus, slot, func,
                         PCI_INTERRUPT_LINE);
uHALr_printf("\tIRQ is @ %d\n", irq) ;
```


7. Initialize the µHAL interrupt subsystem and request control of the interrupts. At this point, if the device generates an interrupt, `tulipInterrupt()` is called.

```
/* init the irq subsystem in uHAL */
uHALr_InitInterrupts() ;
/* assign the interrupt */
uHALr_RequestInterrupt(irq, tulipInterrupt,
  (unsigned char *)"Digital 21142 interrupt handler") ;
```

When the above program is run on a PCI supporting system, the output is similar to the following:

```
ARM Firmware Suite
(c) Copyright ARM Ltd 1999
uHAL v1.1
Initialising...done
Found Digital 21142 ethernet device [00:11:00]
CSRs are at 0x00000000 (IO) and 0x40000000 (Memory)
CSR 00: FE000000
CSR 01: FFFFFFFF
CSR 02: FFFFFFFF
CSR 03: B96998AD
CSR 04: 354F9D62
CSR 05: F0000000
CSR 06: 32000040
CSR 07: F3FE0000
CSR 08: E0000000
CSR 09: FFF483FF
CSR 10: FFFFFFFF
CSR 11: FFFE0000
CSR 12: 000000C6
CSR 13: FFFF0000
CSR 14: FFFFFFFF
IRQ is @ 15
```

This shows that the 21142 was found in slot 11 on bus 0. On this system (an Integrator) this means that the device generates interrupts using bit 15 of the interrupt controller. If it is moved to another PCI slot, it might generate a different interrupt.

9.8 PCI initialization on Integrator

The Integrator system uses a V3 Semiconductor V360EPC to provide PCI host bridge support. The system-specific µHAL code must initialize this device and provide PCI access mechanisms.

9.8.1 Integrator PCI subsystem overview

The V3 PCI interface chip in Integrator provides several windows from local bus memory into the PCI memory areas. Because there are too few windows for our usage, one of the windows is reused for access to PCI configuration space. The memory map is shown in Table 9-2.

Table 9-2 PCI memory map

Local bus memory	Function	Size
`0x40000000` – `0x4FFFFFFF`	PCI memory, non-prefetchable	256M
`0x50000000` – `0x5FFFFFFF`	PCI memory, prefetchable	256M
`0x60000000` – `0x60FFFFFF`	PCI I/O	16M
`0x61000000` – `0x61FFFFFF`	PCI Configuration	16M
`0x62000000`	V3 internal registers	-

There are three V3 windows, each described by a pair of V3 registers. These are:

- `LB_BASE0` and `LB_MAP0`
- `LB_BASE1` and `LB_MAP1`
- `LB_BASE2` and `LB_MAP2`.

Base0 and Base1 can be used for any type of PCI memory access. Base2 can be used either for PCI I/O or for I20 accesses. By default, µHAL uses this only for PCI I/O space.

Note

PCI Memory is mapped so that assigned addresses in PCI Memory match local bus memory addresses.

If a PCI device is assigned address `0x40200000`, that address is a valid local bus address as well as a valid PCI Memory address. PCI I/O addresses are mapped to start at zero. This means that local bus address `0x60000000` maps to PCI I/O address `0x00000000` and so on.

Table 9-3 shows base registers used for mapping the PCI spaces.

Table 9-3 Base register mapping

Local Bus Memory	Purpose	Base/map registers
0x40000000 – 0x4FFFFFFF	Memory	LB_BASE0, LB_MAP0
0x50000000 – 0x5FFFFFFF	Memory	LB_BASE1, LB_MAP1
0x60000000 – 0x60FFFFFF	I/O	LB_BASE2, LB_MAP2
0x61000000 – 0x61FFFFFF	Configuration	

This causes I20 and PCI configuration space accesses to fail. When PCI configuration accesses are required (using the µHAL PCI configuration space primitives) the spaces are remapped as shown in Table 9-4.

Table 9-4 Base register remapping

Local bus memory	Usage	Base/map registers used
0x40000000 – 0x4FFFFFFF	Memory	LB_BASE0, LB_MAP0
0x50000000 – 0x5FFFFFFF	Memory	LB_BASE0, LB_MAP0
0x60000000 – 0x60FFFFFF	I/O	LB_BASE2, LB_MAP2
0x61000000 – 0x61FFFFFF	Configuration	LB_BASE1, LB_MAP1

To make this work, the code depends on overlapping windows working. The V3 chip translates an address by checking its range within each of the BASE/MAP pairs in turn (in ascending register number order). It uses the first matching pair. So, for example, if the same address is mapped by both LB_BASE0/LB_MAP0 and LB_BASE1/LB_MAP1, the V3 uses the translation from LB_BASE0/LB_MAP0.

To allow PCI Configuration space access, the code enlarges the window mapped by LB_BASE0/LB_MAP0 from 256M to 512M. This occludes the windows currently mapped by LB_BASE1/LB_MAP1 so that it can be remapped for use by configuration cycles. At the end of the PCI Configuration space accesses, LB_BASE1/LB_MAP1 is reset to map PCI Memory.

Finally, the window mapped by LB_BASE0/LB_MAP0 is reduced in size from 512M to 256M to reveal the now restored LB_BASE1/LB_MAP1 window

Note

I20 mapping is not set up because using I2O disables most of the mappings into PCI memory.

9.8.2 Initializing the host bridge

The PCI initialization code is an assembly macro in `target.s`. Example 9-3 shows the code.

Example 9-3 PCI initilization code

```
    ; NOTE: load $w1 with the base address of the V3's register set
    ; at the start of the macro and expect it not to change!
      MACRO
$label   SETUP_PCI   $w1, $w2, $w3, $w4

    ; first turn on PCI
    LDR   $w1,  =INTEGRATOR_SC_PCIENABLE
    LDR   $w2,  =0x1
    STR   $w2,  [$w1]
    ; Load up the base address of the V3 register set
    LDR   $w1,  =PCI_V3_BASE

    ; we can NOT try ANY reads from the V3 bridge chip until LB_IO_BASE is written
    ; we ASSUME that we've already waited for >=230us (@PCLK 25MHz) since reset
    ; so that this write WILL have an effect on the V3 chip
    ; Set up where the V3 registers appear in the memory map (PCI_V3_BASE)
    LDR    $w2,  =PCI_V3_BASE
    MOV    $w2,  $w2, LSR #16
    STRH   $w2, [$w1, #V3_LB_IO_BASE]

    ; Wait for the V3 to realise that there is no SROM
    LDR    $w2,  =0xAA    LDR   $w3,  =0x55
30  STRB   $w2,  [$w1, #V3_MAIL_DATA]
    STRB   $w3,  [$w1, #V3_MAIL_DATA + 4]
    LDRB   $w4,  [$w1, #V3_MAIL_DATA]
    CMP    $w4,  #0xAA
    BNEk   %b30
    LDRB   $w4,  [$w1, #V3_MAIL_DATA + 4]
    CMP    $w4,  #0x55
    BNE    %b30
```

```
; Make sure that V3 register access is not locked, if it is, unlock it.
 LDRH   $w2, [$w1, #V3_SYSTEM]
 AND    $w2, $w2, #V3_SYSTEM_M_LOCK
 CMP     $w2, #V3_SYSTEM_M_LOCK
 LDREQ   $w2, =0xA05F
 STREQH  $w2, [$w1, #V3_SYSTEM]

 ; ensure that slave accesses from PCI are DISabled while we set up windows
 LDRH   $w2, [$w1, #V3_PCI_CMD]              ; get current CMD register
 BIC    $w2, $w2, #(V3_COMMAND_M_MEM_EN :OR: V3_COMMAND_M_IO_EN)
 STRH   $w2, [$w1, #V3_PCI_CMD]              ; MEM & IO now BOTH bounce

 ; Clear RST_OUT to 0: keep the PCI bus in reset until we're finished
 LDRH   $w2, [$w1, #V3_SYSTEM]
 BIC    $w2, $w2, #V3_SYSTEM_M_RST_OUT
 STRH   $w2, [$w1, #V3_SYSTEM]

 ; Make all accesses from PCI space retry until we're ready for them
 LDRH   $w2, [$w1, #V3_PCI_CFG]
 ORR    $w2, $w2, #V3_PCI_CFG_M_RETRY_EN
 STRH   $w2, [$w1, #V3_PCI_CFG]

 ; Set up any V3 PCI Configuration Registers that we absolutely have to
 ; LB_CFG controls Local Bus protocol.
 ; enable LocalBus byte strobes for READ accesses too
 LDRH   $w2, [$w1, #V3_LB_CFG]
 ORR    $w2, $w2, #0x0C0                 ; set bit7 BE_IMODE & bit6 BE_OMODE
 STRH   $w2, [$w1, #V3_LB_CFG]

 ; PCI_CMD controls overall PCI operation
 ; enable PCI bus master;
 ;        for memory but NOT I/O
 LDRH    $w2, [$w1, #V3_PCI_CMD]
 ORR     $w2, $w2, #0x04                 ; set bit2 MASTER_EN
 STRH    $w2, [$w1, #V3_PCI_CMD]

 ; PCI_HDR_CFG controls PCI master timeouts etc.
 ; PCI_SUB_VENDOR contains an info field for other masters

 ; PCI_SUB_ID contains an info field for other masters

 ; PCI_MAP0 controls where the PCI to CPU memory window is on the Local Bus
 LDR    $w2, =INTEGRATOR_BOOT_ROM_BASE   ; start of EBI memory
 MOV    $w2, $w2, LSR #20                ; clip to 12-bit field
 MOV   $w2, $w2, LSL #20           ; at top of word wide reg
```

```
; aperture size is 512M
ORR   $w2, $w2, #V3_PCI_MAP_M_ADR_SIZE_512M
; PCI_BASE0 reg MUST be enabled before writing it
; aperture itself enabled too
ORR   $w2, $w2, #V3_PCI_MAP_M_REG_EN :OR: V3_PCI_MAP_M_ENABLE
STR   $w2, [$w1, #V3_PCI_MAP0]        ; finally write the reg
; PCI_BASE0 is the PCI address of the start of the window
LDR   $w2, =INTEGRATOR_BOOT_ROM_BASE  ; 1:1 mapping to start of EBI memory
MOV   $w2, $w2, LSR #20               ; clip to 12-bit field
MOV   $w2, $w2, LSL #20               ; at top of word wide reg
; read may NOT be prefetched for this aperture (MAY change for later FPGA)
; BIC $w2, $w2, #V3_PCI_BASE_M_PREFETCH bit already 0 => NO pre-fetch
STR   $w2, [$w1, #V3_PCI_BASE0]


; PCI_MAP1 is LOCAL address of the start of the window
LDR   $w2, =INTEGRATOR_HDR0_SDRAM_BASE ; start of aliassed header memory
MOV   $w2, $w2, LSR #20               ; clip to 12-bit field
MOV   $w2, $w2, LSL #20               ; at top of word wide reg
; aperture size is 1024M
ORR   $w2, $w2, #V3_PCI_MAP_M_ADR_SIZE_1024M
; PCI_BASE1 reg MUST be enabled before writing it
; aperture itself enabled too
ORR   $w2, $w2, #(V3_PCI_MAP_M_REG_EN :OR: V3_PCI_MAP_M_ENABLE)
STR   $w2, [$w1, #V3_PCI_MAP1]        ; finally write the reg
PCI_BASE1 is the PCI address of the start of the window
LDR   $w2, =INTEGRATOR_HDR0_SDRAM_BASE ; 1:1 mapping to start of header memory
MOV   $w2, $w2, LSR #20               ; clip to 12-bit field
MOV   $w2, $w2, LSL #20               ; at top of word wide reg
; read may NOT be prefetched for this aperture (MAY change for later FPGA)
; BIC $w2, $w2, #V3_PCI_BASE_M_PREFETCH       ;### bit already 0
STR   $w2, [$w1, #V3_PCI_BASE1]
; PCI_INT_CFG controls PCI interrupt pins
; FIFO_CFG controls V3 FIFOs in both directions

; FIFO_PRIORITY controls V3 FIFOs in both directions
; Set up the windows from local bus memory into PCI configuration, I/O
; and Memory
; ... PCI I/O, LB_BASE2 and LB_MAP2 are used exclusively for this
LDR          $w2, =PCI_IO_BASE
MOV          $w2, $w2, LSR #24              ; clip to 8-bit field
MOV          $w2, $w2, LSL #8               ; at top of half-word reg
ORR         $w2, $w2, #V3_LB_BASE_M_ENABLE
STRH  $w2, [$w1, #V3_LB_BASE2]
LDR        $w2, =0            ; map to I/0 address 0 and above
STRH  $w2, [$w1, #V3_LB_MAP2]
```



```
; ...PCI Configuration, use LB_BASE1/LB_MAP1.  Set up on the fly by
;    the PCI Configuration access code in board.c

; ...PCI Memory, use LB_BASE0/LB_MAP0 and LB_BASE1/LB_MAP1
;    Map first 256Mbytes as non-prefetchable via BASE0/MAP0
LDR   $w2, =PCI_MEM_BASE
MOV   $w2, $w2, LSR #20              ; clip to 12-bit field
MOV   $w2, $w2, LSL #20              ; at top of word wide reg
ORR   $w2, $w2, #0x80                ; Window size is 256 Mbytes (7:4 = 1000)
ORR   $w2, $w2, #V3_LB_BASE_M_ENABLE
STR   $w2, [$w1, #V3_LB_BASE0]
LDR   $w2, =PCI_MEM_BASE             ; PCI_MEM_BASE maps to PCI MEM
                                     ; address at  PCI_MEM_BASE
MOV   $w2, $w2, LSR #20              ; clip to 12-bit field
MOV   $w2, $w2, LSL #4               ; at top of half-word reg
ORR   $w2, $w2, #0x0006              ; 3:0 = 0110 = PCI Memory read/write
STRH   $w2, [$w1, #V3_LB_MAP0]
; Map second 256Mbytes as prefetchable via BASE1/MAP1
LDR   $w2, =PCI_MEM_BASE+SZ_256M
MOV   $w2, $w2, LSR #20              ; clip to 12-bit field
MOV   $w2, $w2, LSL #20              ; at top of word wide reg
ORR   $w2, $w2, #0x84                ; Window size is 256 Mbytes
                                     ; 7:4 = 1000), prefetchable
ORR   $w2, $w2, #V3_LB_BASE_M_ENABLE
STR   $w2, [$w1, #V3_LB_BASE1]
LDR   $w2, =PCI_MEM_BASE+SZ_256M
MOV   $w2, $w2, LSR #20              ; clip to 12-bit field
MOV   $w2, $w2, LSL #4               ; at top of half-word reg
LDR   $w2, =0x0006                   ; 3:0 = 0110 = PCI Memory read/write
STRH   $w2, [$w1, #V3_LB_MAP1]

; Allow accesses to PCI Configuration space
; and set up A1,A0 for type 1 config cycles
LDRH   $w2, [$w1, #V3_PCI_CFG]
BIC    $w2, $w2, #V3_PCI_CFG_M_RETRY_EN
BIC    $w2, $w2, #V3_PCI_CFG_M_AD_LOW1  ; force A1=0 and
ORR    $w2, $w2, #V3_PCI_CFG_M_AD_LOW0  ; A0=1 for config type 1
STRH   $w2, [$w1, #V3_PCI_CFG]

; now we can allow in PCI MEMORY accesses
LDRH   $w2, [$w1, #V3_PCI_CMD]          ; get current CMD register
ORR    $w2, $w2, #(V3_COMMAND_M_MEM_EN+V3_COMMAND_M_IO_EN)
STRH   $w2, [$w1, #V3_PCI_CMD]      ; MEM now accepted
                                    ; IO still bounced)
```

```
    ; Set RST_OUT to take the PCI bus is out of reset, PCI devices
    ; can initialise and lock the V3 system register so that no one else
    ; can play with it
    LDRH   $w2, [$w1, #V3_SYSTEM]
    ORR   $w2, $w2, #V3_SYSTEM_M_RST_OUT
    STRH   $w2, [$w1, #V3_SYSTEM]
    ORR   $w2, $w2, #V3_SYSTEM_M_LOCK
    STRH   $w2, [$w1, #V3_SYSTEM]

MEND
```

9.8.3 PCI configuration cycles

The PCI configuration cycle access routines are in the Integrator `board.c` file.

Access macros are defined for reading and writing registers within the V3 device as shown in Example 9-4.

Example 9-4 Defining access macros

```
// V3 access routines
#define _V3Write16(o,v) (*(volatile unsigned short *)(PCI_V3_BASE + \
    (unsigned int)(o)) = (unsigned short)(v))
#define _V3Read16(o)    (*(volatile unsigned short *)(PCI_V3_BASE + \
    (unsigned int)(o)))

#define _V3Write32(o,v) (*(volatile unsigned int *)(PCI_V3_BASE + \
    (unsigned int)(o)) = (unsigned int)(v))
#define _V3Read32(o)    (*(volatile unsigned int *)(PCI_V3_BASE + \
    (unsigned int)(o)))
```


The PCI configuration window is opened and closed as show in Example 9-5. Without these routine calls, PCI Configuration is not accessible.

Example 9-5 Opening and closing the PCI config window

```
void _V3OpenConfigWindow(void) {
   //Set up base0 to see all 512Mbytes of memory space
   //(not prefetchable), this frees up base1 for re-use by
   // configuration memory
   _V3Write32(V3_LB_BASE0,((PCI_MEM_BASE & 0xFFF00000)| 0x90 | \\
                V3_LB_BASE_M_ENABLE)) ;
   //Set up base1 to point into configuration space, note that
   //MAP1 register is set up by uHALir_PCIMakeConfigAddress().
   _V3Write32(V3_LB_BASE1, ((PCI_CONFIG_BASE & 0xFFF00000) | 0x40 | \\
               V3_LB_BASE_M_ENABLE)) ;
}

void _V3CloseConfigWindow(void) {
   //Reassign base1 for use by prefetchable PCI memory
   _V3Write32(V3_LB_BASE1, (((PCI_MEM_BASE + SZ_256M) & 0xFFF00000) | 0x84 | \\
               V3_LB_BASE_M_ENABLE)) ;
   _V3Write16(V3_LB_MAP1, (((PCI_MEM_BASE + SZ_256M) & 0xFFF00000) >> 16) | \\
               0x0006) ;
   // And shrink base0 back to a 256M window (NOTE: MAP0 already correct)
   _V3Write32(V3_LB_BASE0, ((PCI_MEM_BASE & 0xFFF00000) | 0x80 | \\
               (pointer unsigned char)V3_LB_BASE_M_ENABLE)) ;
```

The routine in Example 9-6 is used each time access is made to the PCI Configuration space. This maps the offset into PCI Configuration space into a local bus address. It copes with whether or not the bus is the local bus and also with addresses that have bits A31:A24 set. As a side-effect, this routine might alter the contents of LB_MAP1 so that the V3 can generate the correct addresses.

Example 9-6 Configuration space offset mapping

```
 unsigned int uHALir_PCIMakeConfigAddress(unsigned int bus, unsigned int device,\\
                            unsigned int function, unsigned int offset) {
    unsigned int address, devicebit ;
    unsigned short mapaddress ;

   if (bus == 0) {
    /* local bus segment so need a type 0 config cycle */
    /* build the PCI configuration "address" with one-hot in A31-A11 */
      address = PCI_CONFIG_BASE ;
      address |= ((function & 0x07) << 8) ;
    address |= offset & 0xFF ;
    mapaddress = 0x000A ;                   /* 101=>config cycle, 0=>A1=A0=0 */
    devicebit = (1 << (device + 11)) ;
    if ((devicebit & 0xFF000000) != 0) {
      /* high order bits are handled by the MAP register */
      mapaddress |= (devicebit >>16) ;
    } else {
      /* low order bits handled directly in the address */
      address |= devicebit ;
    } else { /* bus !=0 */
      /* not the local bus segment so need a type 1 config cycle */
      /* A31-A24 are don't care (so clear to 0) */
      mapaddress = 0x000B ;                    /* 101=>config cycle,
                                                  1=>A1&A0 from PCI_CFG */
     address = PCI_CONFIG_BASE ;
      address |= ((bus & 0xFF) <<16) ;       /* bits 23..16 = bus number      */
      address |= ((device & 0x1F) << 11) ;   /* bits 15..11 = device number   */
      address |= ((function & 0x07) << 8) ;   /* bits 10..8  = function number */
     address |= offset & 0xFF ;               /* bits  7..0  = register number */
    }
    _V3Write16(V3_LB_MAP1, mapaddress) ;

    return address ;
}
```


Example 9-7 shows a typical usage of the configuration routines. In this example, a byte is read from PCI Configuration space:

Example 9-7 Reading a byte from PCI Configuration space

```
unsigned char uHALr_PCICfgRead8(unsigned int bus, unsigned int device,\\
                                unsigned int function, unsigned int offset) {
pointer unsigned char  pAddress ;
unsigned char          data ;
    // open the (closed) configuration window from local bus memory
_V3OpenConfigWindow() ;

    /* generate the address of correct configuration space */
    pAddress = (pointer unsigned char)(uHALir_PCIMakeConfigAddress(bus, device, \\
                                                        function, offset)) ;

    /* now that we have valid params, go read the config space data */
    data = *pAddress ;

    // close the window
    _V3CloseConfigWindow() ;

    return(data) ;
}
```

9.8.4 Interrupt routing

The Integrator-specific interrupt routing code uses a static routing table, as shown in Example 9-8. This provides the generic PCI code with mapping of the interrupt numbers, the slot a device occupies, and the interrupt pin it uses.

Example 9-8 Interrupt mapping

```
unsigned char uHALir_PCIMapInterrupt(unsigned char pin, unsigned char slot) {
#define INTA IRQ_PCIINT0
#define INTB IRQ_PCIINT1
#define INTC IRQ_PCIINT2
#define INTD IRQ_PCIINT3

//DANGER! For now this is the SDM interrupt table...
    char irq_tab[12][4] = {
    // INTA  INTB  INTC  INTD
    {  INTA, INTB, INTC, INTD },     // idsel 20, slot 9
    {  INTB, INTC, INTD, INTA },    // idsel 21, slot 10
    {  INTC, INTD, INTA, INTB },    // idsel 22, slot 11
    {  INTD, INTA, INTB, INTC },    // idsel 23, slot 12
    {  INTA, INTB, INTC, INTD },    // idsel 24, slot 13
    {  INTB, INTC, INTD, INTA },    // idsel 25, slot 14
    {  INTC, INTD, INTA, INTB },    // idsel 26, slot 15
    {  INTD, INTA, INTB, INTC },    // idsel 27, slot 16
    {  INTA, INTB, INTC, INTD },    // idsel 28, slot 17
    {  INTB, INTC, INTD, INTA },    // idsel 29, slot 18
    {  INTC, INTD, INTA, INTB },    // idsel 30, slot 19
    {  INTD, INTA, INTB, INTC }     // idsel 31, slot 20
} ;
uHALr_printf("pin = %d, slot = %d\n", pin, slot) ;

    if (pin == 0) pin = 1 ; //if PIN = 0, default to A
    return irq_tab[slot-9][pin-1] ; //return the magic number
}
```

9.9 Rebuilding the PCI library

Use the project files or makefiles to rebuild the PCI Library. See Chapter 11 *Building AFS Components* for more information on rebuilding AFS components.

9.9.1 PC project files

You can build the PCI Library with SDT 2.5 project manager files (`.apj`) or ADS 1.0 CodeWarrior project files (`.mcp`).

9.9.2 Unix makefile

The CD has a makefile for use on a Unix workstation.

There is a makefile for rebuilding the PCI Library for a single development board and processor combination. For example, if you copied `unix\source` contents to `/AFS` use `/AFS/Integrator940T/PCI/Build/makefile` to rebuild the library for the Integrator board with an ARM940T processor.

You must maintain the hierarchy of the CD directories when you copy the files from the CD to your workstation. The makefile defines `ROOT` as the root of the build tree and is needed by `mk`. `TOOLS` is the tools directory that contains build tools of various kinds.

For general information on makefiles and directory structure, see *AFS source structure* on page 11-4.

Chapter 10
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:

10.1 Frequently asked questions

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

10.1.1 Does AFS support Thumb?

The CodeWarrior IDE project files installed for the AFS components and demo applications use a define in the **C Pre-processors** and **Assemblers** 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 which build non-Thumb versions. These builds can be run on any supported processor.

IntegratorT.b

This directory contains project files which 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
```

10.1.2 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
```

Use `make ADS_BUILD=0` to build for SDT. All AFS components can be built for both SDT and ADS without the need for you to edit the source code.

Note

The `scatter`, `ropi`, and `rwpi` build options are not supported.

10.1.3 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
```

APM projects that build applications with the ADS C library require an additional assembler predefine:

```
USE_C_LIBRARY {SETL} TRUE
```

and an additional C pre-processor `#define`:

```
USE_C_LIBRARY=1
```

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 µHAL now defines the heap and stack base and size and exports these to be used by the C library memory management routines. Also, program start-up and termination are controlled by C library routines.

Note

The SDT C libraries are not supported.

10.1.4 Is µHAL free?

The µHAL 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.

10.1.5 Can I use µHAL in my project?

You can use µHAL in your commercial projects, but you must synchronize with ARM the release of µHAL that you are using and the testing procedure.

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.

10.1.6 What boards are supported?

The current set of supported evaluation boards consist of:

- ARM Development Board (PID7T)
- Integrator
- Prospector.

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

If you are using the Integrator board, you can load and debug applications using Multi-ICE:

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 RAM, or use the debugger console to access the flash utilities.

The boot monitor commands are not available from Multi-ICE, but you can use the flash utilities AFU and BootFU to access the flash memory. See Chapter 8 *Using the ARM Flash Utilites*.

10.1.8 How can I verify that Angel is installed?

To test whether Angel is installed on your board:

1. Set the terminal emulator to 9600 baud.
2. Set the configuration switches to boot the Angel image.
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.
5. If you do not see the Angel banner, you must load the Angel image for your board.

10.2 Troubleshooting

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

10.2.1 µHAL does not work with my processor

µHAL supports the following processors:

- ARM7TDMI
- ARM720T
- ARM740T
- ARM920T
- ARM940T
- StrongARM (SA110 and SA1100).

If the processor you want to use is not on this list, you might still be able to use parts of the µHAL source code or definitions in your application.

10.2.2 The boot switcher fails to run an image from Integrator 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.
- Use the 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.

10.2.3 The boot switcher fails to run an image from Prospector 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.
- Check that switch U25-5 is correctly set.

10.2.4 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.

10.2.5 Exception vector errors when using Multi-ICE

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

- On some processors, 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*.

10.2.6 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 has already been Requested. The System Timer ID is available using `uHALir_GetSystemTimer()`.

10.2.7 Enabled timer interval is too long

The interval count is reloaded when the timer is enabled, so it will be the full interval duration (or slightly longer if the system is busy) before the timer event occurs again.

10.2.8 Terminal emulator does not work with boot monitor

The line parameters for the boot monitor are the defaults for the µHAL 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).

10.2.9 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.

10.2.10 Demo applications run slower as standalone

The µHAL 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 by a factor of four giving the performance loss.

Two demos in flash 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. This can be achieved by setting the `Read Only` address for the linker to `0x8000`.

When programming the image into the 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 exploit this information is in the ARM boot monitor.

Chapter 11
Building AFS Components

This chapter describes how to build AFS components. (Use the same process to build the demonstration applications.) Prebuilt images of each component are provided, but you might build a component if you have modified the source code or if you are constructing a different board and processor combination. This chapter contains the following sections:

11.1 AFS component variants

The components can be generic or board-specific:

- Generic components run on all platforms and do not require board or processor-specific files.
- Board-specific components require definition files specific to the board and processor.

For some components you can have a generic version or a board-specific version. If the generic version is built, some features of the target processor are not available since the code is built to run on any ARM processor.

The image resulting from the build is either:

Standalone Standalone images do not require a host workstation. All I/O is processed on the target board using the hardware on the development board. `putc()`, for example, can use the UART port to send characters.

Semihosted Semihosted images require a host workstation. Some I/O, for example file access, is processed on the host computer. A communications link (serial port or Multi-ICE) is used with a debugger on the host computer to interpret and process file access.

Where there are both standalone and semihosted versions of a component, both versions are built from the same makefile or project file.

There are three ways of building the AFS components:

- using ARM `.apj` project files with SDT 2.5
- using ARM `.mcp` project files for the CodeWarrior IDE with ADS 1.0
- using GNU makefiles (Windows and Cygwin `make` or Unix `gnumake`).

11.1.1 Installing the component directories

The ARM Firmware Suite consists of a number of components. By default, each component is held in a subdirectory. If, for example, you copy the `windows\source` contents from the CD into `C:\AFS\source`, the source code and build tools are in the following directory structure:

`C:\AFS\source\all`

For rebuilding AFS components for multiple boards at one time.

`C:\AFS\source\Integrator`

For applications for the Integrator board that are not processor-specific.

`C:\AFS\source\Integrator`*`processor_number`*

For applications for the Integrator board that are specific to the *processor_number* processor.

`C:\AFS\source\Prospector`

For applications for the Prospector board that are not processor-specific.

`C:\AFS\source\Prospector`*`processor_number`*

For applications for the Prospector board that are specific to the *processor_number* processor.

`C:\AFS\source\Pid`*`processor_number`*

For applications for the PID ARM Development Board that are specific to the *processor_number* processor.

11.2 AFS source structure

There are build directories for each development board. For example, it is possible to build µHAL for an Integrator board with either the ARM740T or ARM940T processor. In this case, only the processor-specific code differs.

Each component of AFS has a similar directory structure. The build files use relative paths. The paths given in this section assume that you copied the contents of the `windows\sources` directory from the CD to `C:\AFS` and that you are building components for the Integrator board with a ARM940T processor. The same relative path descriptions apply for the other boards.

The Angel build process is slightly different. See *Building a µHAL-based Angel* on page 6-10 for details on building Angel for your development board.

11.2.1 The Build subdirectory

This directory contains the makefile and project files. How AFS components are built is deliberately separated out from the sources themselves. All build definitions and files are kept in the `Build` directory of each component.

This directory contains a sub-directory per board variant that can be built for this component. The path for the boot monitor project files and makefile, for example, is:

```
C:\AFS\source\Integrator940T\bootMonitor\Build\Integrator940T.b
```

The built images are kept in sub-directories of the board-specific build directory. For example, if you build a standalone version of the µHAL demonstration program `system-timer.c` for a Integrator board fitted with an ARM940T based header card, the output file would be `system-timer.axf` and the path is:

```
C:\AFS\source\Integrator940T\uHALDemos\Build\Integrator940T.b\standalone\
```

11.2.2 The Sources subdirectory

This contains the generic sources for the component. For a pure µHAL component such as the `uHALDemos`, all of the sources can be found here. (A pure application is one that only uses µHAL APIs for system-specific code. An impure application is one that has code that directly accesses the system-specific hardware.) These sources do not include any board or processor-specific code although they might differentiate between semihosted and standalone operation.

11.2.3 The Boards subdirectory

This contains board-specific code for the component. The board-specific code is kept in a separate directory for each board. For example, the path to the board-specific code needed to run µHAL on an Integrator platform is:

```
C:\AFS\source\Integrator940T\uHAL\Boards\INTEGRATOR
```

If a component does not have board-specific code, there is not a `Boards` directory.

The `h` directory

This subdirectory contains the include files. These files contain definitions of routines and structures.

11.2.4 The µHAL directories

The µHAL component has two extra directories that contain board and processor files.

The Processors directory

Each supported processor has specific code for memory management unit and caches. For example, the ARM940T cache flushing code is located in:

```
C:\AFS\source\Integrator940T\uHAL\Processors\ARM940T\mmu940T.s
```

11.3 Using ARM project files

Versions of ARM project files for use with the *ARM Software Development Toolkit* (SDT) version 2.5 and the *ARM Development Suite* (ADS) version 1.0 can be found in the board-specific build directories. These are called `uHALlibrary.apj` for SDT 2.5 and `uHALlibrary.mcp` for ADS 1.0. To build a particular variant, click on the project file and build.

For more information about using ARM project files with SDT 2.5, see the *Software Development Toolkit User Guide*. For more information about using ARM project files with ADS 1.0, see the *ADS CodeWarrior IDE Guide*.

11.3.1 Using CodeWarrior IDE (.mcp) project files

The CodeWarrior IDE project files (`.mcp` extension) are the build files designed for use with ADS. Operation instructions and help are available from the ADS manuals or through the on-line help available within the CodeWarrior IDE.

The build system is initiated by either:

- Using the Host PC point and click interface to select the `.mcp` file.
- Selecting the Codewarrior icon and loading the required project file using **Project→Open** from the Codewarrior IDE Menu.

Either of these methods starts the IDE and makes the required project the focus window.

The two image types (Targets) produced by the CodeWarrior IDE are semihosted and standalone.

The build target can be changed in two ways, by using the pull down menu showing the target name (**semihosted** or **standalone**) or by clicking on one of the Targets tabs (**Files**, **Link Order**, and **Targets**), then selecting the appropriate target (see Figure 11-1).

Figure 11-1 Target

The CodeWarrior IDE project files expect the source files to be in the original firmware release directory. If source files are moved they must be removed from the project file by selecting and deleting them from the **Files** window. This generates a message window requiring confirmation of the removal. You can then add the files to the project using **Project→Add Files**. This also adds the appropriate access path to the selected target. This process must be repeated for both the **Standalone** and **Semihosted** targets (see Figure 11-2).

Figure 11-2 Add files

The CodeWarrior IDE also allows you to directly edit the paths to library and include files (see Figure 11-3).

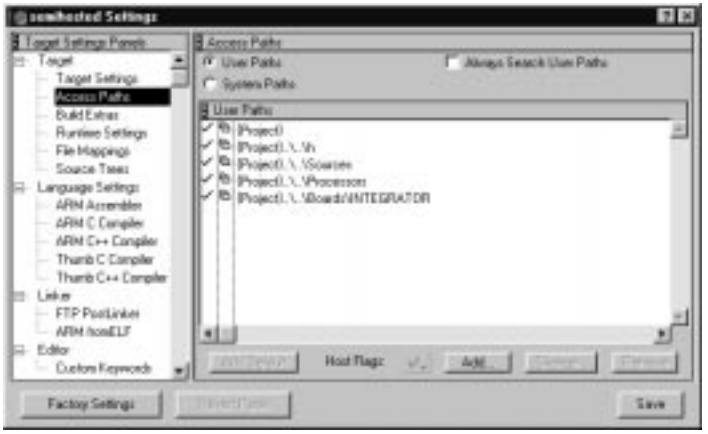

Figure 11-3 Edit paths

To build the chosen target simply click on the **Make** icon, or **Project→Make** (see Figure 11-4).

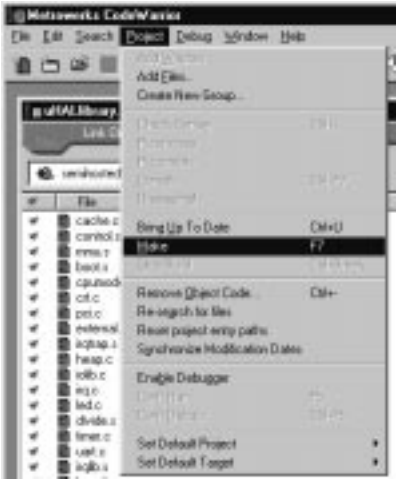

Figure 11-4 Make

The generated image is in the directory:

*project_name*_Data*target_selected*

with the name:

project_name.axf

For example, the µHAL Demo `bubble` when built using the ADS CodeWarrior IDE and a semihosted target generates the image:

```
bubble_Data\semihosted\bubble.axf
```

Note

The ADS CodeWarrior IDE creates a new directory, `uHALLibrary_Data` for example, that contains the `semihosted` and `standalone` subdirectories. Two separate build operations are required to build both the semihosted and standalone versions.

The CodeWarrior IDE does not provide the **Force Build** option provided by the Arm Project Manager. Targets are only rebuilt if any of the **Target Settings** have changed or any of the sources have been modified since the last build. To recreate the **Force Build** option use **Project→Remove Object Code**, then build the target (see Figure 11-5).

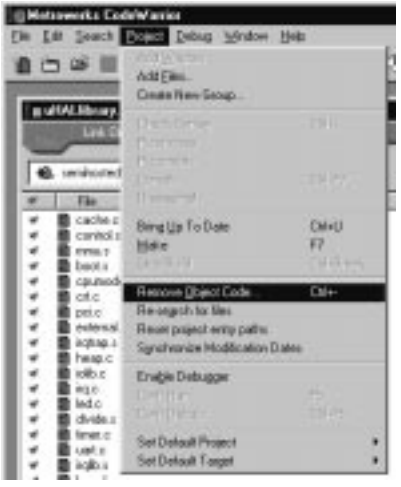

Figure 11-5 Remove

In order to differentiate between SDT and ADS, the build variable ADS_BUILD is defined in both the ARM C Compiler and ARM Assembler panels for the µHAL Library and Demo projects. This ensures that the correct library initialization is carried out. All of the demonstration programs rely on the presence of the µHAL library. If this is not present the µHAL Demo project builds it before attempting to compile any of the demo sources.

You can run the final build image (`.axf`) in the chosen environment.

11.3.2 Using APM (.apj) files

The APM project files (`.apj` extension) are encapsulated build files that contain all the build information. These project files are designed for use within the ARM Project manager environment that is available as part of SDT. Operation instructions and help are available in the ARM SDT manuals.

The build system can be initiated by one of :

- Use the Host PC point and click interface to select the `.apj` file.
- Select the APM icon and loading the required project file using **File→Load** from the APM Menu.

Either of these methods start the APM and make the required project the focus window.

The project files produce one of two selectable image types:

Semihosted Relies on the image being run in a debug environment, such as the ARM Debug monitor or Angel Debug Monitor, to initiate execution of the application and to facilitate text output to a host console. The semihosted image runs from the system RAM at hex address `0x8000`

Standalone Runs on the target system without any additional environment. Text output is achieved by a system port taking a serial stream. The standalone image is linked to the base of the flash memory region of the board. In the case of the ARM Integrator development board, this address is `0x24800000` and requires a utility to program the image into the flash.

The images are selected by expanding the ARM Executable Image heading (select the '+' icon). This shows the two available image options.

The source files are expected to be in the original firmware release directory structure. If the sources need to be moved then they must be removed from the project file and added using **Project→Add Files** to project from the APM menu once in the new directory.

To generate the required image, select the image and select the Force Build option tool button or **Project→Force Build** from the APM menu. The window splits horizontally to show the make procedure. Once built the window shows the project as being up to date. The image will be in the directory of the selected type (semihosted or standalone) with the name *project_name*.axf.

The image is generated as an ARM executable format (.axf) image that can be run from a development board such as Integrator development board.

All of the µHAL Demonstration programs rely on the µHAL library being built. If the library is not built, the project files forces the build. In the event of the PCI Library being required this is also forced. The library builds are controlled as **Sub Projects** within the **Demo** application build.

The final build image (.axf) can be run in the chosen environment.

11.4 Using GNUmake

To build µHAL and its associated components using makefiles use GNUmake. GNUmake is available for UNIX, and for Windows 95 and Windows NT.

Note

Before you can use GNUmake with Windows 95 or Windows NT, you must first install CygWin. GNUmake is available for Windows 95, Windows 98, and Windows NT as part of the Cygwin project. It is recommended that you contact Cygnus at: `http://sourceware.cygnus.com`

11.4.1 Installing GNUmake on Unix

To use GNUmake on a Unix workstation, you must:

- have access to the appropriate versions of the ARM toolset (currently ARM SDT 2.5 or ADS v1.0)
- 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
- place `gnumake` in your search path.

11.4.2 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. Make the environment variable for the ARM C library (ARMLIB) Unix-compliant, for example, `c:/arm250/lib` instead of `c:\arm250\lib`. Without this change, any applications linking against the ARM C library fails to build.
3. Start a bash shell by double-clicking on the desktop shortcut.
4. 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
   ```


5. 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.

6. Add the ARM tools to the bash PATH by editing the `PATH` variable in `cygnus.bat`. For example, change:

```
SET PATH=c:/cygnus/CYGWIN~1/H-I586~1/bin;%PATH%
```

   to:

```
SET PATH =c:/cygnus/CYGWIN~1/H-I586~1/bin;c:/ARM250/Bin;%PATH%
```

7. Test that the ARM tools work by launching a new bash shell and typing `armcc`. `armcc` executes and lists its input options. Alternatively you can edit the path environment variables using the System Properties popup window.

8. 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.)

9. Type the command `gnumake` to build images that contain no debug information, or type `gnumake DEBUG=1` to build images that can be debugged.

11.4.3 makefile locations

The directory you are in when you execute `gnumake` determines how many and what versions of µHAL are built. If, for example, the `windows\source` contents were copied into `C:\AFS\source`, the board and processor version built depends on the directories containing the makefile. For example, three locations are possible:

Build all board and processor combinations

Use the makefile in `C:\AFS\source\all\uHal\` to build versions of µHAL for all boards. The list of available boards is in `all\uHal\boards.in`. The makefile uses a loop to build for all board types:

```
for i in $(BOARDS); \
do (echo '***Making' $$i; \
cd Build/$$i.b ; \
$(MAKE) all) ; \
done
```

Build for a generic Integrator boards

Use the makefile in

`...\source\IntegratorT\uHAL\build`

to build semihosted and standalone versions of µHAL for a generic Integrator board. The code works with the ARM7TDMI and any later ARM processor. The T indicates that this build supports Thumb code. Use `...\source\Integrator\uHAL\build` if you do not want Thumb support.

The contents of `build.in` in this directory are:

```
PROCESSOR_NAME = ARM7TDMI
BOARD_TYPE     = INTEGRATOR
BOARD_NAME     = Integrator940T
```

Build standalone and semihosted targets for Integrator with ARM940T

Use the makefile in:

`...\source\Integrator940T\uHal\Build\Integrator940T.b`

to build the standalone and semihosted versions for the Integrator board with an ARM940T.

The contents of `build.in` in this directory are:

```
PROCESSOR_NAME = ARM940T
BOARD_TYPE     = INTEGRATOR
BOARD_NAME     = Integrator940T
# link specific options
UHAL_PROCESSOR_ARCHITECTURE = arm940t
THUMB_AWARE = 1
```

11.4.4 Build control files

The build is controlled by include files:

Common rules file

The `C:\AFS\source\`*`board`*`_name\uHAL\rules.in` file contains a common set of rules and definitions that are used when building all firmware components for *`board_name`*. The board-specific makefile for each component must include the µHAL rules file.

Board-specific include file

Each board variant supported by µHAL is defined by the file `board.in` in the build directory for that board. For example, the build definition file for the ARM720T variant of the Integrator board is:

`C:\AFS\source\Integrator720T\uHAL\Build\Integrator720T.b\board.in`

This contains:

```
PROCESSOR_TYPE = ARM720T
BOARD_TYPE = INTEGRATOR
BOARD_NAME = Integrator720T
```

These definitions are used in `rules.in` to generate directory names so that the appropriate board-specific and processor-specific modules are assembled, compiled, and included. This allows easy adaptation when new boards are added or removed from the build process.

Environment file

Each component has a top-level file, for example `uHAL\environ.in`, that defines the board variants supported within that component. It also contains relative directory names for µHAL and any other firmware library needed by a particular component.

Generic makefile

The generic makefile `...\uHal\Build\common.make` contains build rules and dependencies for software to boards supported by a particular component. When you add a new µHAL module, `common.make` must be changed. The other board-specific makefiles do not require changing.

µHAL Board Definition File

The following is the `board.in` definition file for the Prospector SA-1100 based board. All of the variables are used by `rules.in` to generate various build flags and definitions:

```
# what flavour board is this?
PROCESSOR_NAME    = SA110
BOARD_TYPE        = PROSPECTOR
BOARD_NAME        = Prospector1100
```

The following variables must be set in the definition file:

BOARD_NAME

The name of this board variant. For Prospector1100, this describes the directory where the built objects are kept. If the build is from:

```
...\source\Prospector1100\uHAL\Build
```

the object files are in:

```
...\source\Prospector1100\uHAL\Build\Prospector1100.b
```

BOARD_TYPE

The type of board. This generates definitions for the compiler and assembler:

```
-DPROSPECTOR=1
-PD "PROSPECTOR SETL {TRUE}"
```

This is also the name of the directory within `uHAL` that contains the board-specific code. If the build is from:

```
...\source\Prospector1100\uHAL\Build
```

the path is:

```
...\source\Prospector1100\uHAL\Boards\PROSPECTOR
```

PROCESSOR_NAME

The name of the processor supported by this board. This describes which directory the processor-specific definitions and macros are taken from. If the build is from:

```
...\source\Prospector1100\uHAL\Build
```

the path is:

```
...\source\Prospector1100\uHAL\Processors\SA1100
```

The included files generate definitions for the compiler and assembler. For Prospector1100 this is:

```
-DSA110=1
-PD "SA110 SETL {TRUE}"
```


The following variables are optional and might be set in the definition file:

UHAL_BOARD_ADEFS

These are board-specific definitions that are included on the assembler command line.

UHAL_BOARD_CDEFS

These are board-specific definitions that are included on the compiler command line.

UHAL_BOARD_LDEFS

These are board-specific definitions that are included on the linker command line.

UHAL_BOARD_ELFDEFS

These are board-specific definitions that are included on the fromELF command line.

UHAL_BOARD_ARDEFS

These are board-specific definitions that are included on the armar command line.

UHAL_BOARD_SOURCES

The set of files that make up the sources for the port of µHAL to this board. This is defaulted to `board.c`, `drver.s`, and `memmap.s`.

UHAL_BOARD_INCLUDES

The set of include files for this board. This defaults to `platform.h`, `platform.s`, and `target.s`.

THUMB_AWARE

If this variable is present and true, AFS builds as ARM-Thumb interworking code. This is the default in some files. The files in `IntegratorT.b`, for example, build Thumb interworking applications.

The environ.in File

This file describes where uHAL is relative to this component and the set of board variants that are built. Example 11-1contains the `environ.in` file for the uHALDemos component of AFS:

Example 11-1 environ.in

```
#
# Copyright (C) ARM Limited 1999. All rights reserved.
#
#-------------------------------------------------------------
# This set of makefile definitions describe the build
# environment for the uHAL based programs
#-------------------------------------------------------------
# Where is uHAL (relative to ./Boards/<BOARD_NAME>)
UHAL_BASE = ../../../uHAL
# Where is PCI library (relative to ./Boards/<BOARD_NAME>)
PCILIB_BASE = ../../../PCI
# Where is the Flash library
FLASHLIB_BASE = ../../../FlashLibrary
# The set of boards/subdirectories that we need to build.
BOARDS = Integrator IntegratorT Integrator720T Integrator740T \
Integrator920T Integrator940T Integrator966T Prospector1100 \
PID7T PID740T PID940T
```

Board-specific makefile

Example 11-2 is the board-specific makefile for the Integrator variant of the uHALDemos component. The environ.in file for this component is used to get the paths to other include and rules files.

Example 11-2 Board-specific makefile

```
# Copyright (C) ARM Limited 1999. All rights reserved.
# uHAL demo makefile
#
include ../../environ.in
#-------------------------------------------------------------
#  Locally defined things.
#-------------------------------------------------------------
BOARD_NAME     = Integrator
```



```
#-------------------------------------------------------------
# Use the uHAL rule sets
#-------------------------------------------------------------
include $(UHAL_BASE)/Build/$(BOARD_NAME).b/board.in
include $(UHAL_BASE)/rules.in

#*************************************************************
#  Make targets
#*************************************************************
#
# On Integrator, we assume that we're running out of Flash at
# 0x24800000 and that we don't care how we got into memory.
# Practically this means being flashed via EmbeddedICE or
# MultiICE.
#
all:
    $(MAKE) TARGET=semihosted semihosted_all
    $(MAKE) TARGET=standalone TEXT=0x24800000 \
    DATA=0x00010000 TYPE='-aifbin' standalone_all

include ../common.make

#*************************************************************
#  clean up the development tree.
#*************************************************************
clean:  # clean up
    $(MAKE) TARGET=semihosted semihosted_clean
    $(MAKE) TARGET=standalone standalone_clean
```

11.4.5 Location of control files

When the makefile requires access to include files located in other directories, the path to the files is defined by relative paths and by defined variables. For example, the makefile in:

```
c:\AFS\source\Integrator940T\uHal\Build\Integrator940T.b
```

includes `rules.in` and `boards.in` from:

```
C:\AFS\source\Integrator940T\uHal\
```

since the UHAL_BASE variable defines a relative path:

```
UHAL_BASE       = ../../
include board.in
include $(UHAL_BASE)/rules.in
include $(UHAL_BASE)/boards.in
```

11.5 Build output files

The ARM Project Manager, CodeWarrior IDE, and GNUmake build ELF files for standalone and semihosted operation. Some of the build tools also make other formats. The formats supported by AFS are:

output.axf

This is an *ARM eXecutable Format* (.axf) file that is an ELF format image. This can be converted into other formats by using the fromELF utility.

output.a

This is an ARM library (armar) format image used with ADS. (This format is used when a library is created that will be linked with other code.)

output.alf

This is an ARM format image used with SDT.

output.aifbin

This is a binary image with an AIF header.

output.bin

This is a plain binary image.

Use the fromELF utility to produce other formats, for example Motorola S-record. For more information on image formats, see the documentation for your ARM toolkit.

11.6 Using the ADS C libraries

This section provides an overview of the *ARM Development Suite* (ADS) libraries. You can extend the library by creating new functions or by replacing existing semihosted functions with standalone functions.

For a detailed description of how the libraries comply with the ISO standard, see the chapter on compilers in the *ADS Tools Guide*.

See the description of semihosting in the *ADS Debug Target Guide* for more information on the debug environment.

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

Building an application with ADS and µHAL combines the high-level support of the C and C++ library with the low-level routines of µHAL.

11.6.1 ADS build options and library variants

When you build your application, you must make certain fundamental choices. For example:

Byte order Big-endian or little-endian.

Processor Different processors have different capabilities, for example Thumb support, that are supported in different library variants.

Scatter loading or single area

µHAL does not support scatter loading.

Floating-point support

FPA, VFP, software, or none. µHAL uses software floating point.

Position-independence

µHAL does not support ROPI or RWPI (position independent code and position independent data).

When you link your assembly language, C, or C++ code, the linker selects appropriate C and C++ library variants compatible with the build options you specified. There is a variant of the ANSI C library for each combination of major build options.

11.6.2 ADS library directory structure

The ADS libraries are installed in two subdirectories within *ADS_install_directory*\lib:

armlib — This subdirectory contains the variants of the ARM C library, the floating-point arithmetic library, and the math library.

cpplib — This subdirectory contains the variants of the Rogue Wave C++ library and supporting C++ functions. The Rogue Wave and supporting C++ functions are collectively referred to as the ARM C++ Libraries.

Note

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

11.6.3 µHAL initialization of the library

Library initialization performed by µHAL during both the standalone and semihosted startup allows µHAL to remain in control of the system. The library initialization routines are:

- `__rt_stackheap_init()`
- `_init_alloc()`
- `_initio()`

An example of initialization code is shown in Example 11-3.

Example 11-3 Library initialization

```
uHALir_CLibInit
    STMFD  sp!, {r14}
    ; Must keep the next two routine calls in order, init_alloc
    ; needs the heap base and limit in r0 and r1 respectively.
    BL   __rt_stackheap_init
    BL   _init_alloc
    LDR   r4, =_initio
    BLNE   _initio      ;Opens stdin, stdout and stderr
    LDMFD  sp!, {pc}    ; restore registers and return
```

Upon branching to main, both the heap and stacks and the standard I/O streams have been initialized. The floating point initialization must also be performed if the image has been built to use software floating point.

11.6.4 µHAL and the ADS run-time memory model

Define the locations of the stack and heap and their respective sizes using the interface provided by the library. µHAL defines a new memory model using the provided library interface.

To define a new memory model `__rt_stack_heap_init()` is provided, along with the stack limit handling function `__rt_stack_overflow()` if the images are to be built with stack checking. The current implementation is shown in Example 11-4.

Example 11-4 Stack and heap initialization

```
__rt_stackheap_init
    LDR  r4, =uHALiv_BaseOfMemory
    LDR  a1, [r4]    ; Here's the top of free RAM
    LDR  r4, =uHALiv_TopOfHeap    ; Top of installed memory
    LDR  a2, [r4]
    MOV pc, lr
```

Since µHAL initializes stacks separately, `__rt_stack_heap_init()` returns only the lower and upper bounds of memory to be used as a heap in `a1` and `a2`. An `_init_alloc()` function then uses this data to initialize library heap management.

The stack pointers for the various processor modes are initialized during µHAL startup. The C library inherits these stacks and therefore no further stack initialization is required. Because there is a valid heap and stack, the library memory management facilities, `malloc()`, `realloc()`, `calloc()`, and `free()` can be used unchanged from the library.

If the heap is fully allocated `__rt_heap_extend()` is called. It attempts to return a pointer to the location of the extended heap. You can use this interface to provide additional non-contiguous blocks of memory to extend the heap. The library does not define a default implementation of this. However, it is used by the memory manager if defined. The current implementation simply returns zero in `a1`, denoting failure.

11.6.5 ADS I/O functions

Each of the I/O functions is based on a SWI interface. Since there is no support within µHAL for the SWI interface in the standalone case, some tailoring of the library is necessary to present a common interface between the standalone and semihosted images for the `printf()` and `scanf()` families. There is no file system for the standalone case. Therefore, file I/O requests must be denied. The I/O support functions require some modification to support standalone images.

11.6.6 ADS trap handling

Any run-time errors found by the library are signalled through the function `__rt_raise()` that in turn calls `__raise()`. If there is no other signal handler available `__default_signal_handler()` is called. This prints a message detailing the error discovered by the library. The message is printed using `_ttywrch()` which uses a SWI interface to output the message one character at a time.

The exception handling interface provided by the library in `_ttywrch()` must be retargeted for the standalone case by the implementation of the required SWI.

The return value from `__raise()` denotes whether or not the exception has been handled and if execution can continue. If the return value is non-zero the program exits through a call to `_sys_exit()` with the return value as the exit code.

11.6.7 Program handling

Library program termination is carried out by a branch to `exit()` that calls `_sys_exit()` to shutdown the library and terminate the program using a SWI interface. Modifications, similar to the library initialization modifications, give µHAL control of program termination.

Therefore a return from `main()` using program startup, as described in *µHAL and the ADS run-time memory model* on page 11-23, results in fall through to `_sys_exit()` within `boot.s`. By shutting the library down from µHAL the present program termination code remains the same with the addition of the following calls to library shutdown routines:

- `_terminateio()`
- `_terminate_user_alloc()`

Example 11-5 shows an example of program termination within µHAL.

Example 11-5 Program termination

```
    BL   main
_sys_exit
    BL  _terminateio  ;close down any file I/O
    BL  _terminate_user_alloc  ;Free any allocated memory
    BL  uHALir_DisableInt
IF :DEF: SEMIHOSTED
    LDR  r0, =angel_SWIreason_ReportException
    LDR  r1, =ADP_Stopped_ApplicationExit
    SWI SWI_Angel
ENDIF
0
    B %0  ; Loop forever
```

11.6.8 Building an application with ADS

This section describes:

- *µHAL with ADS* on page 11-25
- *Build variants* on page 11-26
- *Retargeting µHAL functions* on page 11-27

µHAL with ADS

All µHAL applications, by default, link with the ADS C library. The linker always scans the appropriate C library for any non-weak references that are not resolved by the µHAL library.

If, for example, a µHAL application references the C library `malloc()` functionality, the linker includes the library member that contains `malloc()`, as well as all associated library members and initialization code.

The initialization of the library routine is performed within µHAL startup code. In the example of `malloc()`, the µHAL initialization routine `uHALir_ClibInit()` calls `_init_alloc()` to initialize the library heap allocator. If `malloc()` is not used by the application, the library heap initialization is not included in the image.

Library members, routines (including initilization routines), and definitions are only included in the image if they are directly or indirectly referenced by the application.

Note

There is a single exception. Floating point initialization is only performed if the µHAL library has been built to use software floating point emulation. This is due to the possibility of hardware support that eliminates the requirement for C library floating point initialization code.

Build variants

The build option `USE_C_LIBRARY` controls how a µHAL application links against the ADS C library:

- `USE_C_LIBRARY =0` uses the helper functions provided by µHAL, for example `__rt_udiv()` and `__raise()`.

  There is still some minimal functionality from the ADS C library because the module `rt_memcpy_w.o` is always included from the C library. To build µHAL applications that do not use the C library at all, you must provide your own version of the memcopy routine `__rt_memcpy_w()`. (If you are building an application that uses interworking, you must also provide `__16__rt_memcpy_w()`.)

- `USE_C_LIBRARY =1` uses the helper functions provided by the C library. (Examples of helper functions are `rt_memcpy_w.o`, `sys_command.o`, and `rt_raise.o`.)

If your µHAL application directly references any ADS C library routine, all associated library members are included into the output image and the setting of `USE_C_LIBRARY` is ignored. You can use the `-noscanlib` link option to prevent the including of C library routines. The `-noscanlib` option causes a link failure if non-µHAL library functions are referenced.

Retargeting µHAL functions

If an application references an ADS C library function and there is also a µHAL implementation of the function, the µHAL version of the function is redundant. The µHAL version is retargeted so that any calls are redirected to the C library implementation (see Figure 11-6).

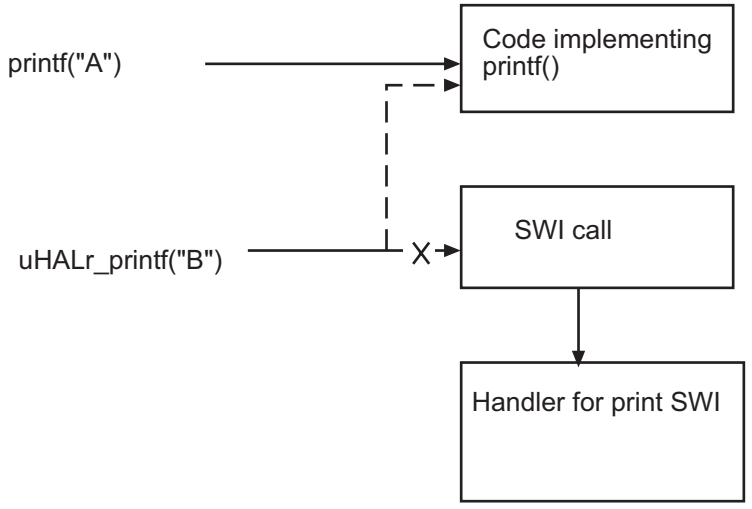

Figure 11-6 Redirection

For example, if an application uses the C library routine `printf()`, any subsequent calls to `uHALr_printf()` are redirected to the C library implementation, `printf()`. This is repeated for `malloc()`, `free()`, `getchar()`, `putchar()`, and `getc()`.

However, if the application links against the C library but makes no reference to non-µHAL functionality, none of the µHAL functions (for example `uHALr_printf()` or `uHALr_malloc()`) are retargeted and appropriate C library initialization is not performed.

Glossary

ADS See *ARM Developer Suite*.

ADU See *ARM Debugger for UNIX*.

ADW See *ARM Debugger for Windows*.

AFU See *ARM Flash Utility*.

AFS See *ARM Firmware Suite*.

ANSI American National Standards Institute.

API See *Application Programming Interface*.

APM See *ARM Project Manager*.

Application Programming Interface The syntax of the functions and procedures within a module or library.

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.

ARM Boot Flash Utility The *ARM Boot Flash Utility* (BootFU) allows modification of the specific boot flash sector on the system.

ARM Debugger for UNIX
The *ARM Debugger for UNIX* (ADU) and *ARM Debugger for Windows* (ADW) are two versions of the same ARM debugger software, running under UNIX or Windows respectively.

ARM Debugger for Windows
See *ARM Debugger for Unix*.

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 eXtendable Debugger
The *ARM eXtendable Debugger* (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 *ARM Flash Utility* (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.

ARM Project Manager
The ARM project manager is the component of SDT that controls building applications. The equivalent in ADT is the CodeWarrior IDE. Unix systems can use makefiles to build applications.

ARMulator
ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.

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.

ATPCS
The *ARM and Thumb Procedure Call Standard* (ATPCS) defines how registers and the stack are used for subroutine calls.

AXD
See *ARM eXtendable Debugger*.

Big-Endian
Memory organization where the least significant byte of a word is at a higher address than the most significant byte.

BootFU
See *ARM Boot Flash Utility*.

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 *Canonical Frame Address*.

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 — *Debug With Arbitrary Record Format* (DWARF) is a format for debug tables.

EC++ — A variant of C++ designed to be used for embedded applications.

ELF — Executable Linkable 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 — Non-volatile 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 — An executable file which has been loaded onto a processor for execution.

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.

	See also Output sections
Input section	Contains code or initialized data or describes a fragment of memory that must be set to zero before the application starts.
	See also Output sections
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 *Big-endian*.
Local	An object that is only accessible to the subroutine that created it.
Load view	The address of regions and sections when the image has been loaded into memory but has not yet started execution.
Memory management unit	Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.
MMU	See *Memory Management Unit*.
Multi-ICE	Multi-processor in-circuit emulator. ARM registered trademark.
Output section	Is a contiguous sequence of input sections that have the same RO, RW, or ZI attributes. The sections are grouped together in larger fragments called regions. The regions will be grouped together into the final executable image.
	See also Region
PCI	See *Peripheral Component Interconnect*.
PCS	Procedure Call Standard.
	See also ATPCS
Peripheral Component Interconnect	An expansion bus used with PCs and workstations.
PIC	Position Independent Code.
	See also ROPI
PID	Position Independent Data *or* the ARM Platform-Independent Development card.

See also RWPI

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 that one instance of the code active. Each instance of the subroutine call has its own copy of any required static data.

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.

Regions In an Image, a region is a contiguous sequence of one to three output sections (RO, RW, and ZI).

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.

See also Input sections

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 *System Information Block*.

System Information Block — A block of user-defined nonvolatile storage.

SWI — Software Interrupt. An instruction that causes the processor to call a programer-specified subroutine. Used by ARM to handle semihosting.

Target — The actual target processor, (real or simulated), on which the application is running.

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.

Thread — 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.

Veneer — 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.

Watchpoint — A location within the image which will be monitored and which will cause execution to break when it changes.

Word — A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

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

B

C

D

E

F

G

H

I

L

M

U

V

Symbols

