digital

Porting Angel and uHAL to a StrongARM Platform

An Application Note

Order Number: EC–RBZLA–TE

March 1998

This application note describes how the uHAL (microHAL) and Angel sources are related. It also recommends a number of coding/porting practices that together allow Angel and uHAL based applications to safely exist.

Revision/Update Information: This is a new document.

Digital Equipment Corporation
Maynard, Massachusetts

http://www.digital.com/semiconductor

March 1998

Contents

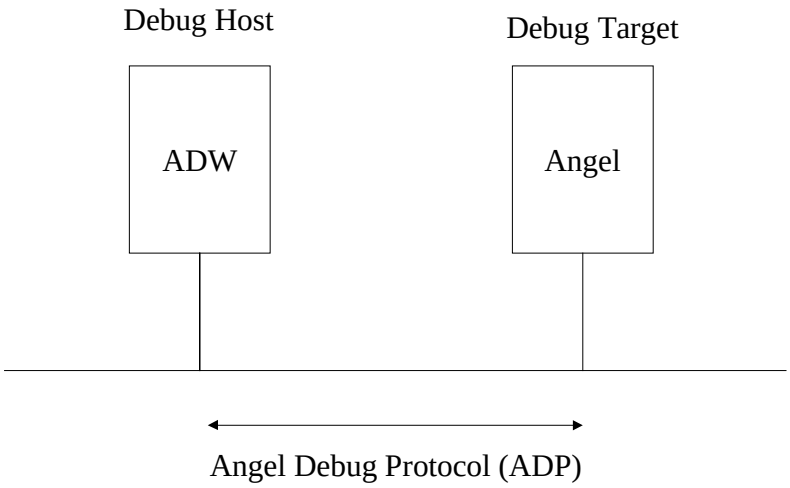
1. Introduction

This document describes how the uHAL (microHAL, an application's portability layer) and the Angel remote debugger sources are related. It recommends a number of coding practices that allow Angel and uHAL to be easily ported to StrongARM based systems and to safely coexist. It also describes the desirable features of a StrongARM board that make software porting and operating system development easier.

The software described is based on the ARM Software Development Toolkit (SDT) Version 2.11 and the uHAL 0.5 release.

2. Background

The following diagram shows an abstract view of the ARM SDT Version 2.11 debug environment. The *debug host*, typically a Windows 95 or Windows NT system, runs the *ARM Debugger* (ADW) application. This Windows application is the host end of the debugger.



The ARM Debugger communicates with the *Angel* debug target[1] running on a StrongARM evaluation board via the *Angel Debug Protocol* (ADP). ADP is a simple command/response protocol that can be carried via a serial line or via the UDP protocol over Ethernet. The media used to carry ADP can be selected from within the ARM Debugger, and a number of StrongARM boards support debugging via Ethernet. At present, these boards are the EBSA-285 and EBSA-110 StrongARM evaluation boards.

[1] ARM's original debug target is known as Demon. Demon is much simpler than Angel and, in many ways, easier to port. However, we recommend that you use Angel; ARM is moving away from supporting Demon.

The Angel debug target manages the StrongARM system and allows the debug host to upload images to be debugged, set breakpoints, examine registers, and so on. Applications written to be loaded, debugged, and run via Angel are known as *Angel images*; they have to be aware that Angel is running and make sure that they behave appropriately. In other words, Angel images must make sure that they do not accidentally stop Angel running correctly. Images that run immediately from power up without the aid of Angel are known as *standalone images*.

ARM releases the sources to the Angel debug target as part of its ARM SDT. These Angel sources are structured so that the board-specific code is separated from the main, board-independent code. DIGITAL ships the board-specific code and prebuilt firmware (Angel images included) with its evaluation board, and also makes them available on its website.

The *bootloader* is a piece of software that runs when the evaluation board boots. Its task is to decide which of several flash ROM-based images is to run and to transfer control of the system to it. The bootloader functions can be combined with Angel on most platforms.

uHAL is a portability library that allows simple applications, such as benchmarks, to run on a number of systems. It provides software examples that can be used by third parties with their tools/boards, or it can be used as a software portability layer. uHAL can be used to build both standalone and Angel images; its API is the same in both cases. A number of real-time operating systems use uHAL. Its advantage is that once an operating system has been ported to uHAL, it will run on all platforms that have had uHAL ported to them.

3. Software Collateral

There are already several ports of Angel to DIGITAL's StrongARM platforms. If you are porting Angel to another StrongARM platform, it is strongly recommended that you take the sources of the system closest to your system. These are:

EBSA-110

The EBSA-110 was the first StrongARM evaluation board built. This SA-110-based system is a large PC form-factor board and is almost obsolete. It contains an Ethernet device so that Angel can be connected via internet protocol (IP) running over Ethernet. This makes downloading software images much faster.

EBSA-285

The EBSA-285 is a PCI-based system (PCI access is via the 21285 support chip). It is the most commonly available StrongARM evaluation board at this time. If you use EBSA-285 with the EBSA-BPL, a passive PCI backplane and a 21x4x based PCI Ethernet card, then you can communicate with Angel via Ethernet.

SA-1100 Evaluation Platform

This is an SA-1100-based verification system. Angel uses the serial port for communication. As yet, Ethernet is not supported; however, it could be supported via a PCMCIA card.

Network Appliance Reference Design

This is a network computer reference design based on the SA-110 processor. It includes an Ethernet device that can be used to communicate with the debug host.

All of these systems run Angel, compiled against the ARM Version 2.11 source tree. DIGITAL maintains source trees for all of these systems. These sources are periodically released to ARM for inclusion in their ARM SDT, and they are also available on our external website (http://www.digital.com/semiconductor/strongarm/strongar.htm). The IP Ethernet stack is common to all boards that support Ethernet, although the Ethernet device drivers vary.

Along with the Angel source trees, DIGITAL also develops and maintains the uHAL sources. It is worth noting that uHAL and Angel share many source files; this includes both definitions and macros for setting up memory, initializing page tables, and so on.

4. Evaluation Board Considerations

It is harder to port Angel to some evaluation boards than to others. In particular, StrongARM processors do not help you much because it is impossible to tell what is going on inside the chip. The following features are important when considering porting Angel to boards.

COM Ports

It is better to have two serial ports as opposed to one serial port; you can use the second port to print debug messages. If you do not have a second serial port, then it is helpful to have another output device, such as an LCD. Remember that you have to get the debug code running in the first place.

LEDs

When you first run Angel on a board, you have no idea of what it is doing. Angel performs a lot of setup before it sends its banner out on the board's serial port. Lighting different LEDs as the code progresses is the only way to know that the startup code has been executed. Therefore, you should consider having a minimum of three LEDs.

Flash ROM

When you are trying to get software working on a system, you need to try the new code quickly. Waiting a long time between trial images makes bringing up the new software slow. Therefore, porting Angel to a new board is easier if you are able to get the Angel image onto the board in a reasonable time. This is particularly important in the early days of a board's life.

Using a ROM emulator makes the transfer of code very fast (as opposed to removing the flash part, programming it, and putting it back). Some ROM emulators support loading via Ethernet, which is very quick; but even loading via a serial line is fairly quick. The ability to download new Angel images quickly allows an engineer porting Angel to try new images quickly, and it considerably reduces the time taken for the overall port. The port of Angel to the SA-1100 board was slowed down considerably by having to remove flash parts from the board and program them on a flash programmer each time a new Angel image was tried.

Programming the flash parts in situ via JTAG is also plausible, although slow. We achieve around 20KB per minute into flash using the JTAG interface on the EBSA-285. The board needs to be designed to allow this. Both EBSA-110 and EBSA-285 support flash programming via JTAG.

We recommend that the board contains flash parts that can be subdivided into a number of logical or physical flash blocks. The flash parts should be large enough to contain more than one image, with the first image containing the bootloader. The bootloader (described in the following section) is the software that runs at reset or power-on and decides which in-flash image should run. These flash parts should also be programmable by software running on the system. When you have decided to be able to boot one of several images from flash, then you will not only need a bootloader, but you will also need a Flash Management utility. For example, the firmware for the EBSA-285 includes an Angel loadable image that uploads executable images from the debug host and programs them into flash.

You also need to consider how you want to get the final software running on your board. When you have Angel running, then you can download software via Angel. You may choose to debug it or simply have it take over the system, using Angel as a simple downloader. This can be slow and cumbersome, particularly if a serial line is used and the image is very large. Using Angel over Ethernet will make this download time much quicker, but you may have to write an Ethernet device driver for your system.

You could also flash the image. If the image is too large for the onboard flash parts, then you could flash a small load program that loads the final image from an externally connected media such as a disk or a PCMCIA flash card. Alternatively, you could start a BOOTP client that uploads an image from a BOOTP server via Ethernet.

4.1 The Bootloader

The bootloader's task is to select which of several images contained in the flash ROM is to be run. Images run this way are known as *standalone images* to distinguish them from images loadable via Angel, which are known as *Angel images*. The bootloader selects the image using an input device, such as jumpers or a selection switch. The EBSA-285 uses a rotary switch whose setting can be read via the X-Bus interface. When the bootloader has determined which image to run, it must find the image, check that it is valid, and then set up the system before transferring control to it.

The current StrongARM evaluation board bootloaders are all built as part of the Angel image. They run at the start of the Angel code at around the time the system's memory map is changed from its initial, boot map to its running map. The code is part of the `STARTUPCODE` macro in the `target.s` module of the Angel sources for the EBSA-110, the EBSA-285, and the SA-1100 evaluation platform. Logically, there is no reason that the bootloader should not be a standalone piece of software that in turn boots either the Angel debug monitor or some other image such as the power-on self-tests for this system.

The runnable images each have a header, which describes the image, appended to the front of them. The first DWORD (32 bit) is a bootloader to the start of the executable image in flash, and you need to take account of the length of the header when linking the image. Thereafter, the header contains a bit mask that describes the flash blocks that the image occupies, an image number, an image name, and a

checksum. Note that the image's number is not necessarily the same as the flash block that it occupies.

The bootloader must determine that the image number selected exists and that it is valid. It finds the image by searching each flash block in turn for an image header and checking that the number of the image matches the number requested. If that image cannot be found, then the default image is run. Normally, this would be image #0, the Angel/bootloader image; however, this is not always the case. The image is valid if the checksum in the header of the image matches the one generated by the bootloader based on the information in the image header. If the selected image is not valid then, again, the default image is run.

Images can either run directly from flash, in which case they will be linked to run from a particular flash block address, or they can run from memory. If the image is to be run from flash, the bootloader simply jumps to the start of the image in flash. If the image is to run from memory, bootloader goes on to initialize the memory and, optionally, set up the memory-management unit (MMU) before copying the image into memory and transferring control to it. Exactly how much of the system has been initialized depends on the particular system. For example, images linked to run out of flash on the SA-1100 board have control transferred to them without the DRAM initialized and without the MMU set up. On the other hand, memory-resident, standalone images on the SA-1100 board have control transferred to them after the DRAM has been initialized and with the full virtual memory mappings enabled.

4.2 The Flash Management Utility

The Flash Management utility is usually (although not always) an image which runs under Angel that can program images into flash in such a way that the bootloader can recognize and boot them. It appends the flash header that the bootloader uses in order to run the image. For convenience, images are named as well as numbered. The Flash Management utility for each board is different because the flash parts tend to be different, but the user interface is very similar. A particular board's FMU sources are held within its firmware base level. Just like Angel, FMU shares files with uHAL; namely `platform.h` and `platform.s`.

The EBSA-110, EBSA-285, and SA-1100 StrongARM evaluation boards all include a Flash Management utility in their firmware, both as sources and as executable images.

5. uHAL

uHAL is a portability library that allows simple applications and operating systems to run on many evaluation boards. uHAL images can be loaded and run via Angel or booted standalone from flash via the bootloader.

It is worth noting that uHAL applications are not linked to the Angel library; they have no knowledge or expectation of Angel's activities, with two exceptions. The first exception is that uHAL avoids sharing resources with Angel; uHAL uses IRQs whereas Angel (at least on StrongARM boards designed by DIGITAL) uses FIQs. The second exception is that, in some circumstances, uHAL uses SWIs to print characters to the debugger console via Angel.

5.1 uHAL Sources

The uHAL source tree is laid out so that board-specific code and definitions are separated from the common code. The include files for a given board can be found in:

`uHAL/h/board/platform.h, uHAL/h/board/platform.s`

> `platform.h` is generated from `platform.s` via an **awk** script. These include files provide both CPU and board-specific definitions.

`uHAL/lib/board/platform.c`

> This file contains most of the platform-specific code. It includes LED support, IRQ handling (masking/unmasking), platform initialization code, and timers.

`uHAL/lib/board/target.s`

> This file contains the board-specific macros that are shared by Angel and uHAL. These include the memory and MMU initialization code. Each macro is commented as to whether or not it is specific to uHAL, to Angel, or to both.

`uHAL/lib/board/driver.s`

> This file contains low-level driver code and MMU page table array definitions.

When uHAL is built, libraries are generated. These libraries are linked to by uHAL applications; for example, a benchmark. uHAL follows the ARM source code convention of placing files in subdirectories whose names reflect their function. For example, the standalone version of the uHAL library for the EBSA-285 board is built in:

`uHAL/lib/ebsa285/standalone/SAlib.alf`

This layout is understood by the makefiles (which are GNU make compatible and recursive) and by the ARM project files. Unfortunately, not all of the board-specific code is outside of the `uHAL/lib` directory:

`uHAL/lib/irq.s, uHAL/lib/cmplxirq.s`

> These files handle interrupt handling, and this code needs to be board specific. The task of the per-board code is to decipher which interrupt occurred. This will be changed in a future release of uHAL.

5.2 Porting uHAL to Another Platform

To port uHAL to another platform, you must first ensure that you have the latest release of uHAL (currently, 0.5). Then you need to add directories, makefiles, and project files for the new board to uHAL. Your aim at this time should be to successfully build one flavor of the library for this particular board. The two possible types of libraries are standalone and Angel.

If Angel has already been ported to the board (and if the shared files were also generated), then building a uHAL image that loads and runs over Angel should

happen very quickly, and we suggest that you try this first because it is easier to debug. You should inherit the following files from Angel:

`/h/<board>/platform.h`

`/h/<board>/platform.s`

`/lib/<board>/target.s`

`/lib/<board>/driver.s`

If Angel has not been ported to the board, you will need to generate the files listed above. To build the uHAL library, you will also need the following files:

`/lib/<board>/platform.c`

`/lib/<board>/makefile`

`/lib/<board>/salib.apj`

Basing your source files on an existing, similar board is a good idea. However, make sure that you fully understand the inherited code.

Next, you should build a simple demonstration program (these are all held in the `uHAL/demo` directory). A good first image is the "hello world" demonstration program `/uHAL/demo/hello.c`. The demonstration images are independent from the board and should build easily.

Getting an Angel image running is relatively easy because you have a debugger and, therefore, can easily examine the progress of the image and make changes where necessary. Getting a standalone image working is a little harder. If you have LEDs on the board, you should consider embedding the LED settings in the startup code for uHAL, `/lib/boot.s`. This allows you to see how you are progressing in the boot process. Until you have a working image, it is inadvisable to enable the processor's caches. If the "hello world" demonstration program manages to run and output to the appropriate serial port, then all well and good. If it does not run, you may be able to use the LEDs to indicate where it has reached or you may have to use a logic analyzer to track the instructions and data that are being executed. A common problem area is the correct setting up of the system's page tables. If you can avoid doing this by running directly out of memory with the MMU disabled, this can help you generate good serial port code. When that is working, you can then go back and get the page tables working.

When you have got the "hello world" program running, you should try to get `tick.c` running. This is a simple program that allocates the system timer and counts the ticks as they happen. The "hello world" program does not turn on interrupts (uHAL uses IRQs), and so this program allows you to test the interrupt handling code for your system. Once `tick.c` runs, the uC/OS demonstration programs should all run.

6. Angel

The following sections describe the Angel sources and how to port Angel to another platform.

6.1 Angel Sources

The Angel sources have the board-specific code within two directories; one directory contains the code:

`angel/angel/<board>`

For example: `angel/angel/brutus`

The other directory contains the makefiles and ARM project files:

`angel/angel/<board>.b/`

For example: `angel/angel/brutus.b`

There are two ways of building Angel: using GNU make and using ARM project files. The makefile is located in `<board>.b/gccsunos/makefile` and the ARM project file is located in `<board>.b/apm/angelsa.apj`.

The board-specific Angel source files are:

`devices.c`

This file describes the number of devices that this particular Angel supports. You could run the Angel Debug Protocol over any or all of the board's serial devices or over any number of Ethernet devices.

`makelo.c`

This file generates assembler symbols for this board, based on the C definitions for this board. During the build of Angel, it generates `<board>.b/lolevel.s`.

`serial.c`

This file contains the serial drivers for this board.

`angel*.h`

This file contains minimal Angel definitions for this board (for example, `angel285.h`). It defines only what is not already defined in `platform.h`. Over time, the definitions in this file will be subsumed into `platform.*` and `angel*.h` will no longer exist.

`banner.h`

This file contains/defines the string that will be reported in the Angel banner when it reboots or is reset. It gets reported in the console window of the remote debugger (and to the command line only debugger). It is extremely helpful if this string includes information about the board, the date that this Angel was built, and whether or not the MMU/caches are enabled.

`devconf.h`

This file describes the capabilities of the system to Angel. It describes the memory layout, and where in memory Angel will put its stacks as well as enabling/disabling a number of board-specific features. You would edit this file to enable/disable PCI support, Ethernet support, and cache support. The file also contains more device definitions that must match `devices.c`.

`platform.h`

This file is shared with uHAL and describes everything about this board. It contains definitions of the memory map, as well as register definitions for the StrongARM and support chip.

`ringsize.h`

This file defines the receive and transmit ring sizes for the serial driver.

`driver.s`

This file contains low-level driver code (for example, to set the LED pattern) and is shared with uHAL.

`mmu_h.s`

This file contains low-level MMU definitions and is shared with uHAL.

`mmumacros.s`

This file contains MMU register twiddling macros and is shared with uHAL.

`platform.s`

This file is the assembly version of `platform.h` and is shared with uHAL.

`target.s`

This file contains the board-specific assembler macros that are used by Angel. This module is included by `startrom.s`, the Angel code that runs at power-on and reset. The main macros are STARTUPCODE, BOOTLD, INIT_RAM, and INIT_MMU.

6.2 Porting Angel to Another Platform

Like uHAL, it is best to port Angel to a similar platform. Most of the files will be the same, but you may need to rework/rewrite the following files:

`platform.h, platform.s`

You need to update all of the board/processor-specific definitions:

`serial.c`

If you are using a previously supported serial port, this is easy. Otherwise, you need to write a new serial driver. In this case, we suggest that you get standalone uHAL working first so that you can debug the basic UART code. If you have a second serial port or an alternative output device (such as the LCD panel on the SA-1100), you can use uHAL to get the debug code working. For example, the first image that we ran on an EBSA-285 was the uHAL LED flashing program; we then used the (working) LED code within Angel to debug that program.

`driver.s`

You may need to rewrite this definition to provide low-level code specific to your board.

`target.s`

The target.s definition requires the most work because this file contains the most important board-specific code. You could get a lot of this board-specific code working in a standalone uHAL image.

When you have what you believe is an Angel image that might work, you then need to get it working on the board. It is a long time (in CPU cycles) between reset and Angel sending its banner out on its serial port. You should also note that Angel's serial driver (at least in StrongARM ports) is interrupt driven. This means that you really need a mechanism for finding out what Angel software has been executed as you hunt for and fix bugs.

One way is not to enable the instruction and data cache and use a logic analyzer to look at memory accesses. Another way is to use LED flashing code to show where the code has reached. If you use LED flashing code, you will end up sprinkling it throughout `startrom.s` and `target.s`. However, be careful what registers you use because the LED flashing code uses some registers to hold temporary values. Also, you will probably need to insert infinite code loops at given points so that you know exactly where the code has reached because, if Angel resets, the LEDs will flash too quickly to read.

When you have got Angel running to the point where the serial driver is being called, then you need some sort of output device to figure out what is happening. If you have only one serial port (for example, the EBSA-285), this can be very difficult.

7. Angel Versus uHAL

uHAL applications need to be able to run with and without Angel. When a uHAL application is loaded and run via Angel, it must not interfere with the normal running of Angel. Typically, uHAL applications want to have interrupts, change the MMU (page tables) and the PSR. Angel does not take kindly to any of these actions.

7.1 FIQs and IRQs

Angel cannot share interrupts; there are no mechanisms for passing off an interrupt from Angel to uHAL or vice versa. Therefore, make sure that Angel and uHAL use disjoint resources; build Angel to use FIQs and uHAL to build IRQs.

7.2 MMU/Page Tables

Angel takes reasonable care to flush caches when it knows that the MMU is enabled. This is because loading instructions as data and then attempting to execute them is not wise on a Harvard architecture processor. Unless you flush the caches at appropriate times, there is a danger that the data cache will contain instructions (just imagine downloading an image file into memory) that you really want to execute.

For example, imagine a scenario where Angel loads one image, runs it, and then loads another image. If the second image loaded is not flushed from the data cache, then Angel will attempt to run the first image because it is still in the

instruction cache. The instruction fetches will not cause the data cache entries to be written because the two caches are completely separate.

If you want to run uHAL images that change the MMU/page tables or play with the caches, you must be aware that Angel is also using those caches. It is a good idea to build Angel with CACHE_SUPPORTED set to TRUE in `devconf.h`; that way, Angel will flush caches when it feels it is appropriate. Meanwhile, you must be careful to do the right thing in your uHAL application. Fortunately, uHAL contains useful code for cache enabling, disabling, and flushing.

7.3 SWIs

On systems where uHAL has a choice, it uses the second COM port for I/O rather than using SWI callbacks to Angel. This is configurable in `platform.h` and `platform.s`.

The danger in making SWI callbacks to Angel is that Angel does not necessarily immediately carry out the action requested. Instead, it queues the work to be done at a later time. This is not a problem as long as Angel can run and service its work queues. Unfortunately, if Angel does not run soon enough, then it will refuse to queue more work. One way to avoid this problem is to not use Angel for I/O; another way is to allow Angel to run by having a periodic timer expire. The timer code, being part of Angel, will cause the Angel scheduler to run.

Support, Products, and Documentation

If you need technical support, a *DIGITAL Semiconductor Product Catalog*, or help deciding which documentation best meets your needs, visit the DIGITAL Semiconductor World Wide Web Internet site:

http://www.digital.com/semiconductor

You can also contact the DIGITAL Semiconductor Information Line or the DIGITAL Semiconductor Customer Technology Center for support.

For documentation and general information:	
DIGITAL Semiconductor Information Line	
United States and Canada:	1–800–332–2717
Outside North America:	1–510–490–4753
Electronic mail address:	semiconductor@digital.com

For technical support:	
DIGITAL Semiconductor Customer Technology Center	
Phone (U.S. and international):	1–978–568–7474
Fax:	1–978–568–6698
Electronic mail address:	ctc@hlo.mts.dec.com

DIGITAL Semiconductor Products

To order the DIGITAL Semiconductor products, contact your local distributor. Evaluation board kits include a complete design kit, Windows NT installation kit, and an accessories kit with an evaluation board.

DIGITAL Semiconductor Documentation

The following table lists some of the available DIGITAL Semiconductor documentation.

Title	Order Number
DIGITAL Semiconductor EBSA-285 Evaluation Board Reference Manual	EC–R6M5B–TE
DIGITAL Semiconductor SA-110 Microprocessor Evaluation Board Reference Manual	EC–QU5KA–TE
DIGITAL Semiconductor 21285 Core Logic for SA-110 Microprocessor Data Sheet	EC–R4CHB–TE
DIGITAL Semiconductor PCI Development Backplane User's Guide	EC–R6M4B–TE