Abstract: This Paper attempts to provide the basic understanding of the BIOS and the Boot loader in general and does a comparative study of the two Boot loaders the Redboot and the Uboot. With its main focus on the hardware abstraction layers that are provided by these bootloaders, which acts as a generic interface between the underlying hardware and the Operating System Software. The paper goes ahead and provides the basic sequences that are followed during the power on startup in each of the boot loaders and the Hardware abstraction layers of the two boot loaders are compared through their interfaces and the standardized modularity through their layers which brings about the compatibility of the boot loaders across all the CPU architectures. In this paper we attempt to study the HAL (Hardware Abstraction Layer) organization for a simple serial device across the two boot loaders. The basic boot up configuration procedure followed for a Beagle Board using the Uboot is also covered as a part of this paper. As a conclusion the paper tries to summarize the need for the HAL and how the standardization of this is achieved in both the boot loaders. The scope of the paper does not cover the need for having different boot loaders with all providing the same services.

Introduction:

BIOS: this stands for the Basic Input and Output System this is a piece of the software that generally resides on the processor chip the main responsibility of this is locate the source of the bootloader that is in which storage device it is available and then give control to the bootloader for its execution. The other functionality includes providing the POST which is a basic test done to check whether all the hardware configured in the system is working as per the requirement. It also sets the system clocks and configures the hardware that would be need to start the process of booting [ref: 1]. BIOS is stored in the Non Volatile Read Only Memory of the chip and it is usage is seldom only during the power on.

Boot loader: It can be seen as a sequence of operation that are done during the startup of the computer system that would facilitate the working of the Operating System. The Boot loader is analogous to Boot Strapping which means getting ready. The main functionality of the bootloader is to perform the Hardware initializations, establishing the connections between the other peripherals and the allocation of the memory in the RAM that would be required by the Operation System and the Applications to run. Bootloaders can be broadly classified as One staged and Multi Staged bootloader. In case of the Multi Staged bootloaders it consists of the primary bootloader whose sole responsibility is to load the secondary bootloader the latter does all the necessary hardware and software initializations. In some embedded Systems does not require a specific bootloader to run it would just executes a set of instructions involving hardware initialization and then it jumps to the application code. More on the Bootloaders in the following Sections..

REDBOOT and the UBOOT are two Bootloaders each providing its own set of services along with the basic boot strapping process.

Redboot: is an Open Source bootstrap Environment that is used as a bootloader and also as a debug environment in the embedded Systems. It uses the eCos HAL layer in order to provide these functionalities. It provides various methods or interfaces over which the software could be downloaded and executed, like the serial and the Ethernet etc. It supports pre and post development activities like the debug during initial stage and flashing during post development. It is completely open source and there are no liabilities considering the intellectual property. It supports a wide variety of architecture it was initially designed for the eCos RTOS and later extended to Linux. It was a very hot topic of interest in the recent years. With the number of embedded Linux devices increasing the idea was to provide a flexible wide range of customization and features under one development environment that is configurable. An additional functionality that is provided by the Red Boot is that in general interrupts are disabled during the process of boot up of the boards, but the Red Boot stands out by providing a Interruptible boot scripts and enables custom boot sequences in which application and data images can be
automatically loaded from flash memory, removable memory cards, or through a network connection. The another advantage of the Redboot is that it could provide customizable POST test that could be used on the for performing the hardware diagnostic routines just before shipping the computer/embedded system to the end user. Another interesting feature of the Redboot is that it supports internalization of the processor and the target state based on the architecture and even for specific handling for the processor.

**Uboot:** The Universal bootloader it is an Open Source boot loader that is presently the most widely used. The reason for this is its compatibility to work for a wide variety of architectures. The Uboot development uses the cross compiler tools chain to build the software in the host x86 computers which are specific to a particular architecture. It is generally used for the embedded System devices and it was initially written for 8xx Power PC architecture and later adapted for others. Debugging using Uboot is possible but it requires the relocation of the boot contents on to the RAM. The Uboot also provides the Initial hardware test during the booting these tests are configurable in the sense that it could be customized for specific hardware. Uboot provides the ability to run shell interpreter that could use to provide commands during the booting process as an interrupt explained above. It also supports script booting wherein the configuration of the booting process could be written in a scripting language like the python and the executed through the shell interpreter. The Basic idea for the development of the Uboot was to provide a most widely applicable bootloader development software that could be used for a range of real time operating systems. Also to mention is the fact that Uboot in general provides most the functionalities that were available in the Redboot. It also supports password protection during the booting and Timeout mechanism during the booting operations.

**HAL:**

Hardware abstraction layer (HAL) an abstraction implemented software (generally assembly level language) between the computer hardware and the operating system level software. It is designed to hide differences in hardware from most of the operating system kernel. The abstraction is intended to address as a software component for most of the operating system kernel, making it modular so that most of the kernel-mode code does not need to be rewritten to run on different systems (different hardware). In other words the HAL can be considered to hold all the drivers for the motherboard components and allows instructions from higher level computer languages (operating system) to communicate with the hardware directly through the HAL. The function of the HAL does not conclude after the boot-up once the Operating system has started up and running the HAL is also responsible for the letting the OS or the applications running at OS level to discover the hardware of the user system (plug and play functionality). HAL could be seen as an software that is responsible for knowing the devices that are available in the system it is also responsible for configuration of those devices and a communication medium that could be used to interface between the different hardwares. The HAL as explained above was designed to provide the software abstraction for UNIX systems. But HAL is being phased out that it is it being integrated onto the udev. The udev functions with HAL as below it sends message through socket to the HAL, and this HAL would perform further device-specific actions. Like for an example, when a new hardware is attached to the system the HAL layer issues an message onto the data Inter process communication bus that would be received by all the software systems which should be notified, this is exactly what happens when a memory flash drive is attached to the system.

An example on how HAL achieves abstraction in Redboot for different architectures

<table>
<thead>
<tr>
<th>ARM PC</th>
<th>Power PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define HAL_ENABLE_INTERRUPTS()</td>
<td>#define HAL_ENABLE_INTERRUPTS()</td>
</tr>
<tr>
<td>asm volatile (</td>
<td>CYG_MACRO_START</td>
</tr>
<tr>
<td>&quot;mrs r3,cpsr;&quot;</td>
<td>cyg_uint32 tmp1, tmp2;</td>
</tr>
<tr>
<td>&quot;bic r3,r3,#0x0C0;&quot;</td>
<td>asm volatile (</td>
</tr>
<tr>
<td>&quot;msr cpsr,r3&quot;</td>
<td>&quot;mfmsr %0;&quot;</td>
</tr>
</tbody>
</table>
HAL Directory organization in RedBoot: In general the HAL layer that is defined in the Redboot can be broadly classified as Platform Dependent and Independent. Another demarcation on the Directory organization is based on the common responsibilities based on the architecture and the variant.

Architectural Hal: This is specific to the Architecture like ARM, PowerPC etc the files present in this folder are responsible for providing the context switching, basic board initialization like the configuration of the general registers of the CPU specific to this architecture and other common hardware interface initialization. It is also responsible for the delivery of the interrupts.

Platform Hal: this layer is implemented above the implementation HAL, this responsible for providing the functions like “board_init”, “watchdog_init “ these are basic functions which are then routed onto the architectural hal based on the target being used. This layer also handles the timer initialization and the input output access interfaces along with the interrupt controllers.

Implementation Hal: This is the layer that lies between the above two layers the responsibility of this layer is to provide abstraction between the architecture and the variants and other On chip devices that are present in the system. Generally the implementation of this HAL is such the boundaries are not clearly defined.

“The above picture is taken from the book EMBEDDED SOFTWARE DEVELOPMENT WITH ECOS by Anthony J. Massa pg no :20 ref no :5 “

The directory organization in the present Hal is such that there are separate directories as above mentioned the
implementation hal is present in the architectural/platform hal depending on the feature's applicability that is whether it is very generic then it is placed in the architectural hal where it contains all the implementation that are specific to the architecture, while the specific requirements of the chip are handled in the platform Hal. If there are features specific to the kernel or the device driver they are placed outside the HAL. However when it comes to the interrupt handling there is a significant merging of responsibilities between the layers. VSR is present in the architectural HAL which hears from the interrupt controller to dispatch the correct ISR. This Interrupt controller is presently implemented in the Implementation HAL.

The second HAL sub module defines the variant in the HAL file subsystem. A variant is the specific processor within the architecture of the processor family described. For example, a feature that might be present at this level is the support for an on-chip peripheral such as a Memory Management Unit (MMU). And finally the third HAL sub module is the platform. The platform will be the specific piece of hardware that will include the selected processor architecture and, possibly, a variant of the same hardware. This module will include the programs for the platform startup, chip select configuration, interrupt controllers, and timer devices.

As and when new platforms and architecture ports are developed, the packages are updated in the appropriate locations in the HAL directory structure. With the provision of new ports are made available at various times, the directory structure will change according to the new additions.

Let’s consider the subdirectory arch, located under every architecture sub folder. The arch subdirectory will have the files for generic support for the processor architecture. Functionality for this generic support will consist of exception data initialization, startup configuration for ROM and RAM, common interrupt and exception handling, thread context switch handling, a generic linker script file, and common debugging functions. Some HAL architectures would include a subdirectory to contain the variant code in case of the variations for the hardware and this subdirectory is named var.

Example HAL Function Call Trace

Let’s consider the function call __reset() function defined in the common subdirectory in the HAL. For the example function call we will consider the MIPS Atlas evaluation board as the target hardware platform. The sub modules that implement the functionality of the reset function may vary for different HAL architectures.

A description for the steps in the function call is as follows

1. The __reset() routine is a common reset function for all of the HAL packages. The source code for this routine is found in the hal_stub.c file which will be under the common\current\src subdirectory.
2. hal_atlas_reset() routine is called next, and its defined in plf_misc.c located in mips\atlas\current\src subdirectory for the MIPS Atlas platform.
3. Finally, the platform reset routine will use the architecture macro which is defined in plf_io.h located in mips\atlas\current\include subdirectory to toggle the appropriate register within the processor. The HAL_REG() macro will write to the MIPS Atlas reset register.

Uboot Directory organization:

In general the Uboot directory can be broadly classified as the diagram mentioned below
Some of the important directory folders in the uboot:

**board**: this folder contains the information specific to the target board. It provides interface functions like the “board_init”, “hw_init” which would in turn call the specific registers of the target to configure the hardware.

**common**: this folder contains the commands lists that are sent from the shell to the uboot code during the bootup process.

**cpu**: This folder contains the architecture specific folders like the arm, powepe “however this folder is renamed to arch” in the recent version of the uboot. This is similar to the architectural hal present in the redboot.

Drivers: this is the folder containing all the low-level drivers for the system, specifically for a particular target. For eg: atmel_usart.c is the atmel specific USART configuration file.

**rtc**: this folder contains the real-time clock information for all the real-time clocks.

**include**: this directory is specific to the architecture and target:

- **include/config/<boardname>.h**: this is the file that contains the basic configuration for memory map and peripherals. It contains the chip configuration, NAND flash, SDRAM, memory, serial configuration, and much more.

- **include/<cpu.h>**: this contains the particular CPU register for the file. For eg: ‘mpc5xx.h’ contains the registers specific to the CPU “#define SYPCR_BMT 0x0000ff00” is the configuration port for the bus monitoring unit for the mpc (powerPC) target.

Now let us look in the startup Sequences of the two bootloaders.

**RedBoot StartUp Sequence**:

The startup process deals with the initialization of the hardware, the many types of sub-modules of the HAL look after the different initialization process, such as coordination of the operation of the ROM monitor, invoking static and C++ constructors, and start of the application code.
In the coming paragraphs we discuss the hardware initialization and the boot up for the PowerPC-based Motorola MBX860 development board. The startup procedures may differ from different architectures but as far as initialization, when certain steps are completed, the rest of the startup remains pretty much the same.

The detailed description of the startup process is discussed below.

1. The starting point for system startup is after a power up has occurred, this can also imply a soft reset startup of the system and we shall call it the Hardware Powerup.

2. Following a hard or soft reset, the processor starts from its reset vector or a start address (called reset_vector in general). This reset vector is generally found in the file vectors.s in the arch subdirectory for all of the HAL architectures. This file in all of the packages will contain the starting point for that particular HAL package. The responsibility of the reset_vector is to configure/initialize the processor registers to allow the system for a startup process.

3. The reset vector then jumps to _start; also found in vectors.s and is the common starting point of HAL for all system initialization.

4. The following call is the call to the routine hal_cpu_init, which can be located in variant.inc or arch.inc based on the architecture selected. The function of this call is to handle or set registers of the processor of that particular architecture, then to disable the instruction and data caches, to start up the processor in a known state for rest of the initialization process.

5. The next call is the hal_hardware_init following the hal_cpu_init. The function of this call is specified as contained in this routine since the routine is platform specific and can be found in the platform assembly file (for the Motorola MBX board, this is the mbx.s file). In this routine the hardware is set up which includes cache configuration, initialize the interrupt registers to a default state, setting real-time clock registers, disable the processor watchdog, and to configure selected registers on the chip depending on the platform/hardware.

6. The next step, startup routine sets up an interrupt stack area. So making some reserve a storage area to save state information of the processor on the occurrence of an interrupt. We can configure the amount of space to be reserved in the common configuration component. The startup context temporarily uses the interrupt stack to perform its initialization; to make calls into C routines as stack structure for the c programs. And interrupts are disabled during the startup procedure and this will not create a conflict.

7. The following function call is the hal_mon_init function, which is located in the file variant.inc or platform.inc, and this depends on the configuration. This function when running as a ROM application, the function of this task is for the routine to ensure that all default exception handlers for every exception condition are installed and are supported by the processor.

8. The next step in the initialization process, the BSS section is cleared, which will generally contain all uninitialized local and global variables with static storage class.

9. For the C function calls the stack is then set up and can be made from within the vectors.s assembly code.

10. The hal_platform_init routine is called, which is located in hal_aux.c file, which again is based on specific platforms. This in turn, calls hal_if_init, which is located in the file hal_if.c of the HAL common subdirectory. hal_if_init initializes the virtual vector table which is based on the configuration options selected.

11. Next is to initialize the MMU (Memory management unit), which deals with the translations of logical addresses to physical addresses at the same time providing protection and caching mechanisms. The routine hal_MMU_init is located in the file hal_misc.c where the file is located under the arch subdirectory.

12. The next step is to enable the data and instruction cache and this is done in hal_enable_caches, which is found in the file hal_misc.c in the arch subdirectory that particular processor.

13. This follows the routine hal_IRQ_init in order to set up the Communications Processor Module (CPM), where the priorities of the internal and external interrupts are set up. This particular step is specific to the PowerPC processor and can be found in the file hal_intr.c in the arch subdirectory.
14. Then all of the global C++ constructors are called from cyg_hal_invoke_constructors. This routine is located in the file hal_misc.c in the arch subdirectory and the linker handles the generation of the list of global constructors. The file cyg_type.h, located in the infra subdirectory, actually contains macros that define the order in which these constructors are called.

15. When a configuration is set up for a debug environment and a ROM monitor is not providing debug support, the routine initialize_stub is called. This is located in the HAL common subdirectory in the file generic_stub.c initialize_stub, which will install the standard trap handlers and initialize the hardware for debugging.

16. The last step in the HAL initialization process is to hand over the control over to the booted kernel and its initialization. The routine cyg_start takes care of the smooth HAL-to-kernel transition. And based on the lower level hardware initialization the routines in the kernel will take care of the kernel initialization process.

These steps sum up the initialization of the HAL in the RedBoot and following this application are given generalized API to the underlying hardware.

Uboot Startup Sequence:
The Startup sequence using the uboot is explained for the beagle board that uses an OMAP 3 processor, the booting process involved here is generally multi-staged and the which can be broadly classified into three

**ROM code**: this is the on-chip BIOS upon power on it initializes the clocks and minimal peripheral configurations and searches the available devices for the X-loader this X-loader is the primary boot-loader which would in-turn call the uboot for the configuration. The ROM Code analyses the SYSBOOT Pins which would direct the ROM code to the exact location of the X-loader upon finding it the ROM code copies the contents of the X-loader onto the SRAM that is present in the beagle board.

**X-loader**: This is the primary boot-loader and its main objective is to initialize the SDRAM and store the uboot onto it upon which the uboot shall go ahead and invoke the kernel. The first command that is executed upon reaching the SRAM is in the file (/arch/arm/cpu/armv7) start. S_start routine. The user is set in the privilege mode called the SVC32 mode this is used to access the auxiliary control registers.

The next sets of instructions that are followed is to initialize the hardware allocate the basic SDRAM with the necessary size of the uboot and then invoke the uboot, the “cpu_init_cp15” is called herein all the icache and mmu are invalidated then the next routine that is called is the “cpu_init_crit” which internally creates a temporary stack area in the SRAM that is required to handle the stack operations for the presently executing code.

Here are some of the important functions that are being called and their actions

setup_clocks_for_console(): initialises all the UART ports necessary to communicate through the console “do_io_settings()”: this function actually sets the serial device settings, “prcm_init();” //initializes all the clock functions, “imer_init();” //loads the timer with the initial value 0 and starts the timers “sdram_init();” //this function further initialises the sdram.

After the above execution the Flash the Xloader now jumps to the SRAM and executes the following “init_baudrate” the baud rate is set and then the serial communication is set up by the “serial_init”, then the Xloader does few hardware configurations such as the “init_func_i2c” it is here that the the i2c is initialized and the functionality is handled in the SRAM (not in the SDRAM), this is the part of the primary bootloader requirement.

**Uboot**: After doing these operations the code is branched to the flash from where in the following operations are handled the .bss section is cleared the instruction cache is invalidated the memory unit is initialized the interrupts are enabled also incase the watchdog is not initialized that part is handled in the uboot, the nand is initialized and the “stdioinit()” is called this initializes the serial.i2c and the finally the ethernet is initialized and this finally calls the function do_bootm_linux() which would call the main function of the linux kernel.

Serial Device Organization Redboot and Uboot:
serial Device in Redboot :

The Serial device is accessed from the user level in the following way:

The folder in the redboot “io/serial/common/” contains the file “serial.c” this is the core file of the serial driver it is here that an device table is created with the name specified in the file for eg:

DEVI0_TABLE(cyg_io_serial_devio, serial_write, serial_read, serial_select, serial_get_config, serial_set_config) ,These are the user access function that the user would call in order to access the particular functionality of the serial device ,the DEVI0_TABLE is a macro that would create an table specified by the name “cyg_io_serial_devio”. These operations form a part of the highlevel driver generally called the serial core.

The low level drivers are generally defined in the architecture specific files : mpc8xxx_serial.c and .h in the location “serial/powerpc/mpc8xxx/v3.0” here they get associated with “cyg_io_serial_devio” in by creating a DEVTAB_ENTRY entry along with all the function that are specific to handle serial port by accessing the register data. The structure “mpc8xxx_sxx_serial_channel_smc1” contains the list of the functions like putc and getc which would be called in the end upon the invocation of the user defined function call.

When the DEVTAB_ENTRY is created the function calls the __DEV_TAB an array location specific to the array in the data segment it is here that an device entry is created. Once this is achieved the device is then handled by calling the function “cyg_io_lookup” this is similar to our open function in the user face when this function called with the name defined by the macro “CYGDAT_IO_SERIAL_POWERPC_MPC8XXX_SMC1_NAME” the function gets the control of the file and upon the invocation of the user functions like the “cyg_io_write(handle, test_string, &len)” the “putc” function defined in the file mpc8xxx_serial.c is called and this would write down a character byte directly on to a particular register as required.

The Serial device in the Uboot:

The uboot handling of the serial device in the Uboot is similar to that of the Redboot here the high level drivers called the cores maintains an register for all the devices are present in the low level drivers similarly they also provide the user with the interfaces functions that it needs in order to access the particular serial device.

Both the user function call and the serial device are associated with each other in the serial core by the name.

The low level driver register itself onto the serial core as device by calling the function “serial_register(struct serial_device *dev)” it contains a list of functions that are defined are mapped onto the linked list with each specific function written in the lower level driver to a function pointer in the core. The structure type for the “static struct serial_device mpc5xx_serial_drv = { .name = "mpc5xxserial", .start = mpc5xx_serial_init, .stop = NULL, .setbrg = mpc5xx_serial_setbrg, .putc = mpc5xx_serial_putc, .puts = default_serial_puts, .getc = mpc5xx_serial_getc, .tstc = mpc5xx_serial_tstc, .. }” When the user who wants to write on to a particular port calls the function “serial_assign” with the name of the low level driver the function returns a handler function that would be used to access the function that are defined in the lowlevel driver which provide the possibility for it to access the serial port configuration. when the user function calls “serial_write” this function would in turn call the “mpc5xx_serial_setbrg, putc” through which the serial_write functionality is achieved.

Basic comparison of the HAL of Redboot and the Uboot based on the serial device.

<table>
<thead>
<tr>
<th></th>
<th>RedBoot</th>
<th>Uboot</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.Interface functions in C/C++</td>
<td>1. Interface functions in C99</td>
<td></td>
</tr>
<tr>
<td>2.Contains high-level driver or a core that has a direct interface to the user and the low level driver</td>
<td>2. Contains the Serial core, provides the interface functions to the user and the low-level drivers</td>
<td></td>
</tr>
<tr>
<td>3. The high level creates a table for having all the serial devices possible</td>
<td>3. The low level driver registers itself as a serial device</td>
<td></td>
</tr>
<tr>
<td>4. The low level attach themselves through the DEVTAB entry in to the table</td>
<td>4. The user open the file using open close command and calls the high level functions</td>
<td></td>
</tr>
</tbody>
</table>
5. The LLD provides the interface functions `getc()` and `putc()`

6. The user call `cyg_io_lookup` `serial_put()` and `serial_getc()`

**Conclusion:**
Based on the analysis of both the Bootloaders we could find some similarity in their handling of the abstraction layers, modularity and the base idea on which these Boot loaders were built upon to cater multiple architecture and multiple targets.

We could see that uboot approach in handling the device was more generic and the code traversal to determine the code flow was found to be very easy while on the other hand the Redboot approach for accessing a particular device was more complex and incidentally the code traversal to determine the code flow was very difficult.

The amount of documentation available for the uboot was though greater in number it predominantly focused on the configuration and very less on their architecture upon which it was built. While on the other hand the ecos had an extensive documentation both on configuring and on its architecture and the code flow.

Further to this an attempt was made to flash the Angstrom Embedded OS on the beagle board with the uboot as it base bootloader. The configuration and the necessary binary were generated on line using the on line angstrom build generator provided [ref no :8].The output files are provided along with this paper.

**Acronyms:**

- CPU : Central Processing Unit
- BIOS : Basic Input and Output System
- HAL : Hardware Abstraction Layer
- POST : Power On Self-tests
- RAM : Read Only Memory
- REDBOOT : RedHat Embedded Debug and Boot- Loader
- UBOOT : Universal Bootloader
- eCos : Embedded Configurable Operating System
- .bss : block segment Symbol

**References:**

2. [http://sourceware.org/redboot/](http://sourceware.org/redboot/)
4. [http://ecos.sourceware.org/docs-1.3.1/ref/ecos-ref.b.html](http://ecos.sourceware.org/docs-1.3.1/ref/ecos-ref.b.html)
5. [csecsk1.webs.com/ESD.pdf](http://csecsk1.webs.com/ESD.pdf)