REDBOOT AND U-BOOT

A Comparative Study

Jeevan Prasath
Tejas Krishna
What is a Boot-loader

• A small piece of code responsible for

  • Loading the image of the operating system from the network or a non volatile storage

  • Or fetches a secondary complex boot program
    • Eg. GRUB from disk (in the case of multiple OS), which in turn loads the kernel.

• This piece of code is usually about 512 bytes of an hard disk
Why is it required?

• Hardware initialization

• Boot loader is responsible for the placement of the kernel into main memory and its execution applications some times.

• For the operating system to identify the hardware and all the other peripherals.

• The secondary boot-loaders are generally capable reading many file systems and load the selected OS
# BIOS and BootLoader

<table>
<thead>
<tr>
<th>BIOS</th>
<th>Boot-Loader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Basic I/O system is the first step of the boot process</td>
<td>1. Runs after BIOS and loads the OS Image</td>
</tr>
<tr>
<td>2. It also provides interfaces or abstractions for peripheral devices</td>
<td>2. Does the H/W initialization</td>
</tr>
<tr>
<td>3. Stored in read only permanent memory located at the bottom end of the system memory</td>
<td>3. Allocates necessary space in the RAM</td>
</tr>
<tr>
<td>4. Runs the peripheral POST</td>
<td>4. Can also be of multiple stages</td>
</tr>
<tr>
<td>5. Flashed into the hardware</td>
<td>5. Customizable</td>
</tr>
</tbody>
</table>
The Boot Process

- POST (power on self-test) Pre boot sequence

- Master Boot Record (MBR), on PowerOn the computer first executes a small program stored in Read Only Memory (ROM) along with a small amount of required data
Boot Loaders on Embedded arch.

- Embedded boards booting is generally board dependent
- They have a NAND flash or a NOR flash

- In the case of NOR flash the boot-loader is directly accessed and run from the flash

- In the case of NAND it will generally have integrated boot code in the ROM that loads the code from the NAND flash to a static RAM

- Vendor Documentation generally provides all the details
Hardware Abstraction Layer

• What is it?
  • Implemented in software and lies between the hardware and the OS software
    • Can be assembly or C/C++

• The main functionality is to hide the differences between the many types of hardware and its components from the OS
  • i.e. It provides a API by itself regardless of the underlying hardware
  • Its now being merged as a part of udev
RedBoot

• Its an acronym for RedHat Embedded Debug and Bootloader

• Its an open source boot-loader for embedded systems often and is architecture specific

• It’s a complete bootstrap environment for embedded systems and based on the eCos Hardware Abstraction Layer
Architectures supported by RedBoot

- All major architectures are supported RedBoot
  - ARM
  - PowerPC (MBX860)
  - x86
  - ColdFire
  - MIPS
  - SuperH
  - XScale
RedBoot Functionalities

• It can be used as standardized bootstrap and debug environment on commercial development boards and on an OEM's own prototype hardware

• During the manufacturing process it can provide POST and built in self tests of the hardware

• And finally on deployment it can provide
  1. Configuration of RedBoot based on target
  2. Easy portability to firmware
  3. Kernel upgrades
HAL in Redboot

- Hardware Power-up  // Switch On
- Reset_vector      // Start Address
- _start            // Starting point of the code exec.
- hal_cpu_init      // Starts CPU Defined state
  // Disable Cache and instructions
- hal_hardware_init // includes cache configuration, setting interrupt registers to a default state, disabling the processor watchdog and Setting real clock registers
- Setup interrupt stack // reserves a storage area for saving processor state information when an interrupt arrives
- hal_mon_init      // to ensure that default exception handlers are installed for every exception condition
HAL contd.

- Clear block start symbol (bss) section // Contains all uninitialized local and global variables with static storage class
- Setup C function call stack // Stack setup for the C functions
- `hal_platform_init` // Particular h/w initialization
- `hal_MMU_init` // Translation of logical to physical address and enable cache mechanism
- `hal_enable_caches` // Enable data and instruction Cache
- `hal_IRQ_init` // Sets up CPM which prioritizes internal and external Int.
HAL contd.

- cyg_hal_invoke_constructors // global C++ constructors are called
- initialize_stub // Initializes hardware for debug
- cyg_start // HAL to kernel --On to kernel startup...
Serial Device Organization

- Serial.c io/serial/common/serial.c
- Serial.h // High level drivers
- DEVIO_TABLE(cyg_io_serial_devio, /*Type defined in devtab.h*/
  serial_write, /* User access Functions */
  serial_read,
  serial_select,
  serial_get_config,
  serial_set_config
 ); // Table having Device entries

- serial_write(cyg_io_handle_t handle, const void *__buf, cyg_uint32 *len) {cyg_devtab_entry_t *t = (cyg_devtab_entry_t *)handle;
  serial_channel *chan = (serial_channel *)t->priv;
  serial_funs *funs = chan->funs;
  ((funs->putc)(chan, *buf) == false) }
Serial Device Organization Contd.

• Location: mpc8xxx_serial.c
  mpc8xxx_serial.h

• DEVTAB_ENTRY(mpc8xxx_smc_serial_io_smc1,
  CYGDAT_IO_SERIAL_POWERPC_MPC8XXX_SMC1_NAME,
  0, // Does not depend on a lower level interface
  &cyg_io_serial_devio,
  mpc8xxx_sxx_serial_init,
  mpc8xxx_sxx_serial_lookup, // Serial driver may need initializing
  &mpc8xxx_sxx_serial_channel_smc1
); // Low level driver(h/w specific) be linked to the High level driver
Contd.

• struct serial_channel {
  serial_funs *funs;  // Contains the getc() and the putc() functions
  serial_callbacks_t *callbacks;
  void *dev_priv; // Whatever is needed by actual device routines
  cyg_serial_info_t config; // Current configuration
  ...
};

• mpc8xxx_sxx_serial_flush >
  HAL_DCACHE_FLUSH(txbd>buffer, smc_chan->txsize);
    // Is sent as output
Serial Device Organization

- Basic User interface
- cyg_io_lookup // This is analogous to our ‘open’ function
  - Takes two parameters handler and File location
  - Here the devtab entry \_DEVTAB\_ is iterated recursively and by comparing the name the corresponding handle cyg_io_handle_t is returned

- cyg_io_write(handle, test_string, &len);
  - // Has the handler of the lookup, this handler is nothing but the serial write function defined in the high level driver
U Boot

- Open source Universal Bootloader used in Embedded Devices
- It can built on x86 computer for its supporting architectures
- It's built using a cross development GNU toolchain
- It provides basic capability of flashing the non volatile memory
- Also provides a shell command interpreter
- Originally Written for 8xx PowerPC and later deployed for other architectures
Architectures Supported

- ARM (Beagle Board)
- PowerPC
- MIPS
- X86
- Sandbox
- OpenRisc
U Boot Directory Structure

U-Boot Functionalities

- Functions Supported
  - Initialize and test the hardware
  - Download and run OS and application code.
  - Run test applications directly
  - Can even pass through many file systems
  - Interesting facts at the end of the presentation
Start Up Sequence For Beagle Board

• ROM Code
  • The ROM code can boot from Serial port, SD Card, eMMC NAND etc
  • The ROM code checks the system BOOT (GPIO pins) to determine the valid the X Loader, Once found it loads it into the S-RAM of the chip

• X Loader (Primary Boot Loader)
• U Boot – loader (Secondary Boot Loader)
• Kernel Image
X Loader

- Starts at the reset vector `_start`
- `cpu_init_cpu15` // icache and mmu are invalidated
- `cpu_init_crit` // Temporary stack is init
- `s_init_call` // Watch dog is deactivated UART ports are enabled timer initialized
- `sdram_init();` // Initializes SDRAM and allocates the size of the SDRAM required
- `init_boot_params();` // the device type to load the UBOOT is stored in the data section
X Loader Contd..

- `board_init_f` // Initializes the baud rate serial comm and console, I2C communication is established
- `dram_init` // memory for the UBoot is allocated in the SDRAM
- `_relocation` // From start.c saves SDRAM location BSS is cleared
- `.u_boot_list` // Contents of Uboot is moved to SD RAM, then jump to SDRAM
- `jump_2_ram`: // Invalidate Cache, move to UBoot
U Boot

- `board_init_r_` // Cache enabled Nand, MMC, interrupts
  - init Enable Interrupts I2C, SerialIO, driver System init
- `serial_initialize();`
- `nand_init();`
- `mmc_initialize(gd->bd);`
- `dataflash_print_info();`
- `enable_interrupts();`

- Kernel is Invoked
Serial Device Organization

- Serial.c and serial.h // location.. drivers/serial ->serialcore(High level drivers)
- serial_register(struct serial_device *dev) (called from the target low level driver to register it self as a Serial Device)
- Interfaces provided
  - serial_putchar
  - serial_getchar
  - serial_putchar
  - serial_setbrg
Serial Device Organization

- Low level driver serial.c
  - Location: arch/powerpc/cpu/'target folder'
- serial_register(struct serial_device *dev)

```c
static struct serial_device mpc5xx_serial_drv = {
    .name = "mpc5xx_serial",
    .start = mpc5xx_serial_init,
    .stop = NULL,
    .setbrg = mpc5xx_serial_setbrg,
    .putc = mpc5xx_serial_putchar,
    .puts = default_serial_puts,
    .getc = mpc5xx_serial_getchar,
    .tstc = mpc5xx_serial_testchar,
};
```
• From the User level the invocation is through the commands in the console
• It could also be done by our open/close commands as handled in Linux, which indirectly invokes the console to send the commands during the boot mode
• When ever a command is entered through the shell the corresponding do_'command' function is invoked this in turn calls the serial_putchar or the serial_getc
## Serial Dev Comparison

<table>
<thead>
<tr>
<th>RedBoot</th>
<th>U-Boot</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Interface functions in C/C++</td>
<td>1. Interface functions in C99</td>
</tr>
<tr>
<td>2. Contains high-level driver or a core that has a direct interface</td>
<td>2. Contains the Serial core, provides the interface functions to the</td>
</tr>
<tr>
<td>to the user and the low level driver</td>
<td>user and the low-level drivers</td>
</tr>
<tr>
<td>3. The high level creates a table for having all the serial devices</td>
<td>3. The low level driver registers itself as a serial device</td>
</tr>
<tr>
<td>possible</td>
<td></td>
</tr>
<tr>
<td>4. The low level attach themselves through the DEVTAB entry in to the</td>
<td>4. The user open the file using open close command and calls the high</td>
</tr>
<tr>
<td>table</td>
<td>level functions serial_put() and serial_getc()</td>
</tr>
<tr>
<td>5. The LLD provides the interface functions getc() and putc()</td>
<td></td>
</tr>
<tr>
<td>6. The user call cyg_io_lookup</td>
<td></td>
</tr>
</tbody>
</table>
## Comparison

<table>
<thead>
<tr>
<th>RedBoot</th>
<th>U-Boot</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Can boot from: Sec. HDD, floppy USB, LAN</td>
<td>1. Same as Redboot except ZIP drives</td>
</tr>
<tr>
<td>3. Scripts and advanced commands at boot up</td>
<td>3. Same as Reboot</td>
</tr>
<tr>
<td>5. Supported OS: eCos Linux</td>
<td>5. Supported OS: Vxworks, Solaris, Linux, OpenBSD many more</td>
</tr>
<tr>
<td>6. Executable ELF</td>
<td>6. ELF, UBOOT image format</td>
</tr>
<tr>
<td>7. Supported protocol TFTP, Serial</td>
<td>7. Supported protocol: TFTP serial and NFS</td>
</tr>
</tbody>
</table>
Thank you!

Questions?