Embedded Systems Programming
OS Linux - Bootloaders

Iwona Kochańska

Gdansk University of Technology
Elements of embedded Linux

- **Toolchain**: consists of the compiler and other tools needed to create code for target device.
- **Bootloader**: necessary to initialize the board and to load and boot the Linux kernel.
- **Kernel**: heart of the system, managing system resources and interfacing with hardware.
- **Root filesystem**: contains the libraries and programs that are run once the kernel has completed its initialization.
Starting up embedded Linux

http://en.wikipedia.org/wiki/Linux_startup_process
What is a bootloader?

**Bootloader** - part that starts the system up and loads the operating system kernel.

In an embedded Linux system, the bootloader has two main jobs:

- basic system initialization
- loading of the kernel
What is a bootloader?

When the first lines of bootloader code are executed, following power-on or a reset, the system is in a very minimal state.

▶ DRAM controller not set up so main memory is not accessible,
▶ other interfaces not configured so storage accessed via NAND flash controllers, MMC controllers, and so on, are also not usable.
▶ The only resources operational at the beginning are a single CPU core and some on-chip static memory.
▶ As a result, system bootstrap consists of several phases of code, each bringing more of the system into operation.
What is a bootloader?

- The final act of the bootloader is to load the kernel into RAM and create an execution environment for it.
- The details of the interface between the bootloader and the kernel are architecture-specific but, in all cases, it means:
  - passing a pointer to information about the hardware that the bootloader knows about
  - passing a kernel command line, which is an ASCII string containing essential information for Linux.
- Once the kernel has begun executing, the bootloader is no longer needed and all the memory it was using can be reclaimed.
What is a bootloader?

- Subsidiary job of the bootloader is to provide a maintenance mode for:
  - updating boot configurations,
  - loading new boot images into memory
  - running diagnostics.
Boot sequence - old days

- Bootloader placed in non-volatile memory at the reset vector of the processor.
- NOR flash memory mapped directly into the address space
- On the diagram - bootloader is linked so that there is a jump instruction at that location that points to the start of the bootloader code
In the absence of reliable external memory, the code that runs immediately after a reset or power-on has to be stored on-chip in the SoC; this is known as ROM code.

- It is programmed into the chip when it is manufactured.

- The only RAM that the ROM code has access to is the small amount of static RAM (SRAM) found in most SoC designs (4kB - few hundred kB).
Boot sequence. Phase 1: ROM code

The ROM code is capable of loading a small chunk of code from one of several preprogrammed locations into the SRAM. Locations can be:

- first few pages of NAND flash memory,
- flash memory connected through SPI (Serial Peripheral Interface),
- first sectors of an MMC device (SD card),
- file named MLO on the first partition of an MMC device.
- a byte stream from Ethernet, USB, or UART;
Boot sequence. Phase 2: SPL

- In SoCs where the SRAM is not large enough to load a full bootloader like U-Boot, there has to be a **secondary program loader (SPL)**.

- The SPL must set up the memory controller and other essential parts of the system to loading the (TPL) into main memory (DRAM).

- The functionality of the SPL is limited by its size.

- It can read a program from a list of storage devices, as can the ROM code, using preprogrammed offsets from the start of a flash device, or a known file name such as u-boot.bin.
Boot sequence. Phase 3: TPL

- Full bootloader like U-Boot or Barebox.
- Simple command-line user interface - maintenance tasks such as loading new boot and kernel images into flash storage, loading and booting a kernel.
- At the end of the third phase, there is a kernel in memory, waiting to be started.
- Embedded bootloaders usually disappear from memory once the kernel is running and perform no further part in the operation of the system.
Booting with UEFI firmware

Most embedded PC designs and some ARM designs have firmware based on the Universal Extensible Firmware Interface (UEFI) standard (http://www.uefi.org)

Phase 1: The processor loads the UEFI boot manager firmware from flash memory. May allow user interaction through a text-based or graphical interface.

Phase 2: The boot manager loads the boot firmware from the EFI System Partition (ESP) or a hard disk or SSD (or from a network server via PXE boot). The partition format is FAT32.

Phase 3: Bootloader that is capable of loading a Linux kernel and an optional RAM disk into memory
Booting with UEFI firmware

- The third stage bootloader (TPL) should be in a file named:
  `<efi_system_partition>/boot/boot<machine_type_short_name>.efi`
- For example, the file path to the loader on an x86_64 system is:
  `/efi/boot/bootx64.efi`
Booting with UEFI firmware

Common choices for TPL:

- **GRUB 2**
  - GNU Grand Unified Bootloader, version 2,
  - Most commonly used Linux loader on PC platforms.
  - Licensed under GPL v3 - incompatible with secure booting since the license requires the boot keys to be supplied with the code.
  - [https://www.gnu.org/software/grub/](https://www.gnu.org/software/grub/).

- **gummiboot**
  - Simple UEFI-compatible bootloader which has since been integrated into systemd,
  - Licensed under LGPL v2.1
When the bootloader passes control to the kernel it has to pass some basic information to the kernel

- On PowerPC and ARM architectures: a number unique to the type of the SoC
- Basic details of the hardware detected so far (size and location of the physical RAM, CPU clock speed)
- Kernel command line
  - plain ASCII string which controls the behavior of Linux, setting, for example, the device that contains the root filesystem.
  - It is common to provide the root filesystem as a RAM disk, in which case it is the responsibility of the bootloader to load the RAM disk image into memory.
- [Optionally] Location and size of a device tree binary
- [Optionally] Location and size of an initial RAM disk
Introducing device trees

**Device tree** - a flexible way to define the hardware components of a computer system.

- Usually loaded by the bootloader and passed to the kernel, although it is possible to bundle the device tree with the kernel image itself to cater for bootloaders that are not capable of handling them separately.

- The format is derived from **OpenBoot** (Sun Microsystems) which was formalized as the **Open Firmware** specification, IEEE standard IEEE1275-1994. It was used in PowerPC-based Macintosh computers and so was a logical choice for the PowerPC Linux port. Since then, it has been adapted on a large scale by the many ARM Linux implementations and, to a lesser extent, by MIPS, MicroBlaze, ARC, and other architectures.

http://devicetree.org for more information
Device tree basics

- The Linux kernel contains a large number of device tree source files in:
  
  `arch/$ARCH/boot/dts`

- There are also a smaller number of sources in the U-boot source code in:
  
  `arch/$ARCH/dts`

- The device tree represents a computer system as a collection of components joined together in a hierarchy, like a tree.
Device tree

```dts-v1;
/
{
    model = "TI AM335x BeagleBone";
    compatible = "ti,am33xx";
    #address-cells = <1>;
    #size-cells = <1>;
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a8";
            device_type = "cpu";
            reg = <0>;
        }
    };
    memory@0x80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>; /* 512 MB */
    };
};
```

- begins with a root node, represented by a ‘/’
- subsequent nodes represent the hardware of the system
- each node has a name and contains a number of properties in the form name = "value".
The memory and CPU nodes have a **reg property**, which refers to a range of units in a register space.

- Consists of two values representing the start address and the size (length) of the range.
- Both are written down as zero or more 32-bit integers (cells)
The reg property

- When the address or size values cannot be represented in 32 bits (for example, on a device with 64-bit addressing), two cells for each are needed:

```plaintext
/ {
    #address-cells = <2>;
    #size-cells = <2>;
    memory@80000000 {
        device_type = "memory";
        reg = <0x00000000 0x80000000 0 0x80000000>;
    }
}
```

- The information about the number of cells required is held in `#address-cells` and `#size_cells` declarations in an ancestor node.
**The reg property**

**cpu nodes:**
- In a quad core device they might be addressed as 0, 1, 2, and 3.
- Can be thought of as a one-dimensional array without any depth so the size is zero. Therefore: \texttt{#address-cells = <1>} and \texttt{#size-cells = <0>} in the cpus node, and in the child node, \texttt{cpu@0: node reg = <0>}. 
Phandles and interrupts

**Phandle** - connection of a system component in the device tree
  - to an interrupt controller,
  - to a clock source
  - to a voltage regulator.
Phandles and interrupts

```
/dts-v1/;
{
  intc: interrupt-controller@48200000 {
    compatible = "ti,am33xx-intc";
    interrupt-controller;
    #interrupt-cells = <1>;
    reg = <0x48200000 0x1000>;
  },
  serial@44e09000 {
    compatible = "ti,omap3-uart";
    ti,hwmods = "uart1";
    clock-frequency = <48000000>;
    reg = <0x44e09000 0x2000>;
    interrupt-parent = <&intc>;
    interrupts = <72>;
  },
};
```

- **interrupt-controller node**, #interrupt-cells - how many 4-byte values are needed to represent an interrupt line.
- **serial node**, interrupt-parent - references the interrupt-controller it is connected to by using its label (phandle).
Device tree - include files

Common sections of the device tree can be splitted out into include files, usually with the extension .dtsi.

Open Firmware standard:

```
#include /"vexpress-v2m.dtsi"
```

Borrowed from C:

```
#include "am33xx.dtsi"
```

All of this is resolved if the device tree sources are built using kernel kbuild, which first runs them through the C pre-processor, cpp, where the #include and #define statements are processed into plain text that is suitable for the device tree compiler.
Compiling a device tree

- dtc - the device tree compiler
- The bootloader and kernel require a binary representation of the device tree
- The result is a file ending with .dtb, which is referred to as a device tree binary or a device tree blob
- Compile a simple device tree (without include files):

  $ dtc simpledts-1.dts -o simpledts-1.dtb
  DTC: dts->dts on file "simpledts-1.dts"
## Bootloaders

<table>
<thead>
<tr>
<th>Name</th>
<th>Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Das U-Boot</td>
<td>ARM, Blackfin, MIPS, PowerPC, SH</td>
</tr>
<tr>
<td>Barebox</td>
<td>ARM, Blackfin, MIPS, PowerPC</td>
</tr>
<tr>
<td>GRUB 2</td>
<td>X86, X86_64</td>
</tr>
<tr>
<td>RedBoot</td>
<td>ARM, MIPS, PowerPC, SH</td>
</tr>
<tr>
<td>CFE</td>
<td>Broadcom MIPS</td>
</tr>
<tr>
<td>YAMON</td>
<td>MIPS</td>
</tr>
</tbody>
</table>
Das U-Boot

- began life as an open source bootloader for embedded PowerPC boards.
- then, it was ported to ARM-based boards and later to other architectures, including MIPS, SH, and x86.
- it is hosted and maintained by Denx Software Engineering.
Building U-Boot

- Getting the source code:
  
  ```
  $ git clone git://git.denx.de/u-boot.git
  $ cd u-boot
  $ git checkout v2015.07
  ```

- `configs/` directory - more than 1,000 configuration files for common development boards and devices

- Example:
  
  ```
  $ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
  am335x_boneblack_defconfig
  $ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
  ```
Building U-Boot

The results of the compilation are:

- **u-boot**: U-Boot in ELF object format, suitable for use with a debugger
- **u-boot.map**: symbol table
- **u-boot.bin**: U-Boot in raw binary format, suitable for running on your device
- **u-boot.img**: u-boot.bin with a U-Boot header added, suitable for uploading to a running copy of U-Boot
- **u-boot.srec**: This is U-Boot in Motorola srec format, suitable for transferring over a serial connection
- **MLO**: Secondary Program Loader (SPL).
Installing U-Boot

- If the board has a hardware debug interface, such as JTAG, it is usually possible to load a copy of U-Boot directly into RAM and set it running.
- Some SoC designs have a boot ROM built in which can be used to read boot code from various external sources (such as SD cards, serial interfaces, or USBs)
Installing U-Boot

▶ How to load U-Boot via the micro-SD card:
  ▶ format a micro-SD card so that the first partition is in FAT32 format, and mark it as bootable.

$ sudo mkfs.vfat -F 16 -n boot /dev/mmcblk0p1

▶ Direct SD slot available - card appears as /dev/mmcblk0, otherwise, if you are using a memory card reader, it will be seen as /dev/sdb, or /dev/sdc, and so on.
▶ mount the partition and copy U-Boot and the SPL to it:

$ sudo mount /dev/mmcblk0p1 /media/sdcard
$ cp MLO u-boot.img /media/sdcard/boot

▶ unmount the card.
U-Boot doesn’t have a filesystem. Instead, it tags blocks of information with a 64-byte header so that it can track the contents.

**mkimage** command - prepare files for U-Boot

Example - prepare a kernel image for an ARM processor:

```
$ mkimage -A arm -O linux -T kernel -C gzip -a 0x8000 -e 0x80008000 -n 'Linux' -d zImage ulimage
```
Loading images

- U-Boot doesn’t have a filesystem. Instead, it tags blocks of information with a 64-byte header so that it can track the contents.
- **mkimage** command - prepare files for U-Boot
- Example - prepare a kernel image for an ARM processor:
  
  ```
  $ mkimage -A arm -O linux -T kernel -C gzip -a 0x80008000 -e 0x80008000 -n 'Linux' -d zImage uImage
  ```