Embedded Linux Design & Development

By

Kishore Kumar Boddu

Content

1. **Introduction to Embedded Linux**
    - What is Embedded Linux?
    - Embedded hardware
    - Embedded Software
2. **Study AM335X SoC and BBB Board**
    - Read BBB SRM & TRM (AM335X & BBB Block Diagram)
    - AM335X Initialization/Boot sequence
    - AM335X Memory Mapping
3. **Debian 10 Prebuilt Images porting on KM-BBB.**
4. **Setup Embedded Linux Development Environment**
    - Install Required Packages.
    - Setup U-boot Development Environment.
    - Setup Kernel Development Environment.
    - Setup RFS Development Environment.

Content

5. Boot Loader

- u-boot boot loader overview
- u-boot commands and environment variables
- u-boot Source code Layout
- u-boot Development Environment (Configuration & Compilation)
- u-boot Source code Initialization
- u-boot Customization

6. How to communicate Hardware Registers ?

- How to read sysboot control register from u-boot command prompt?
- Mux/Pad Configuration
- Read Control ID register from u-boot command prompt.

7. U-boot Framework

- How reset command works in u-boot?

What is Embedded Linux?

Embedded Linux is the usage of the Linux kernel and various open-source components in embedded systems

Embedded Linux

Embedded hardware for Linux systems?

Embedded Linux Hardware - Architecture

- The Linux kernel and most other architecture-dependent component support a wide range of 32 and 64 bits architectures
  - x86 and x86-64, as found on PC platforms, but also embedded
  - systems (multimedia, industrial)
  - ARM, with hundreds of different SoC (multimedia, industrial)
  - PowerPC (mainly real-time, industrial applications)
  - MIPS (mainly networking applications)
  - SuperH (mainly set top box and multimedia applications)
  - Blackfin (DSP architecture)
  - Microblaze (soft-core for Xilinx FPGA)
  - Coldfire, SCore, Tile, Xtensa, Cris, FRV, AVR32, M32R

Embedded Linux Hardware - MMU

- Both MMU and no-MMU architectures are supported, even though no-MMU architectures have a few limitations.
- Linux is not designed for small microcontrollers.
- Besides the toolchain, the bootloader and the kernel, all other components are generally architecture-independent

Embedded Linux Hardware – Communication Protocols

The Linux kernel has support for many common communication busses

- I2C
- SPI
- CAN
- 1-wire
- SDIO
- USB
- And also extensive networking support
- Ethernet, Wi-Fi, Bluetooth, CAN, etc.
- IPv4, IPv6, TCP, UDP, SCTP, DCCP, etc.
- Firewalling, advanced routing, multicast.

Types of hardware platforms

- **Evaluation platforms** from the SoC vendor. Usually expensive, but many peripherals are built-in. Generally unsuitable for real products.
- **Component on Module,** a small board with only CPU/RAM/flash and a few other core components, with connectors to access all other peripherals. Can be used to build end products for small to medium quantities.
- **Community development platforms,** a new trend to make a particular SoC popular and easily available. Those are ready-to-use and low cost, but usually have less peripherals than evaluation platforms. To some extent, can also be used for real products.
- **Custom platform.** Schematics for evaluation boards or development platforms are more and more commonly freely available, making it easier to develop custom platforms.

Embedded Linux

Embedded Software Development

Software components

- **Cross-compilation toolchain**
  - Compiler that runs on the development machine, but generates code for the target
- **Bootloader**
  - Started by the hardware, responsible for basic initialization, loading and executing the kernel
- **Linux Kernel**
  - Contains the process and memory management, network stack, device drivers and provides services to userspace applications
- **C library**
  - The interface between the kernel and the userspace applications
- **Libraries and applications**
  - Third-party or in-house

Cross-compiling toolchains

- The usual development tools available on a GNU/Linux workstation is a native toolchain
- This toolchain runs on your workstation and generates code for your workstation, usually x86
- For embedded system development, it is usually impossible or not interesting to use a native toolchain
- The target is too restricted in terms of storage and/or memory
- The target is very slow compared to your workstation
- You may not want to install all development tools on your target.
- Therefore, cross-compiling toolchains are generally used. They run on your workstation but generate code for your target.

Machines in build procedures

- Three machines must be distinguished when discussing toolchain creation
- The **build machine**, where the toolchain is built. (linaro community)
- The **host machine**, where the toolchain will be executed.
- The **target machine**, where the binaries created by the toolchain are executed.
- Four common build types are possible for toolchains

Machines in build procedures

Native build

used to build the normal gcc
of a workstation

Cross-native build

used to build a toolchain that runs on your
target and generates binaries for the target

Cross build

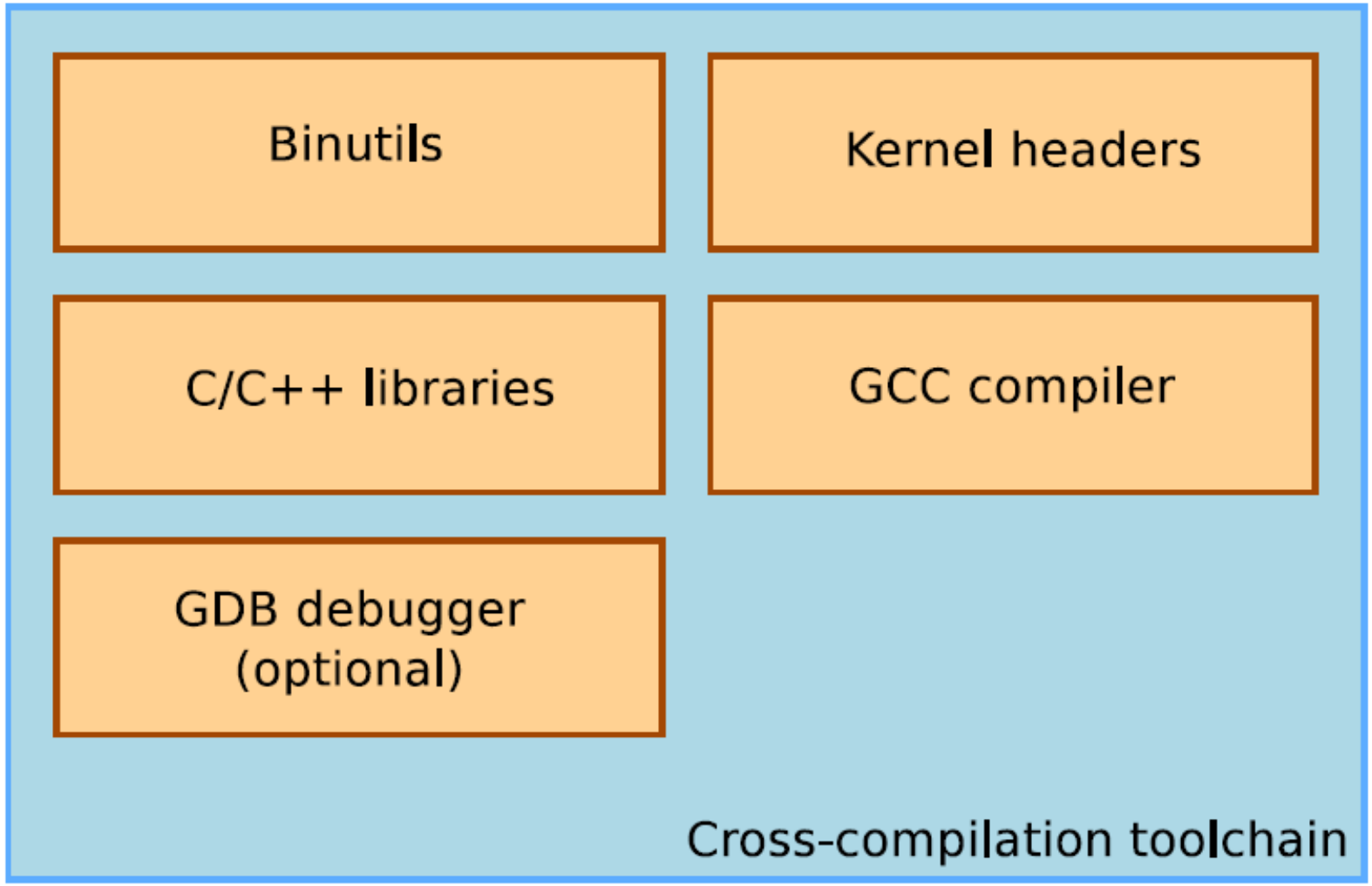
used to build a toolchain that runs
on your workstation but generates
binaries for the target

The most common case in embedded development

Canadian build

used to build on architecture A a
toolchain that runs on architecture B
and generates binaries for architecture C

Components


Binutils
Kernel headers
C/C++ libraries
GCC compiler
GDB debugger
(optional)
Cross-compilation toolchain

Binutils

- Binutils is a set of tools to generate and manipulate binaries for a given CPU architecture
  - as, the assembler, that generates binary code from assembler source code
  - ld, the linker
  - ar, ranlib, to generate .a archives, used for libraries
  - objdump, readelf, size, nm, strings, to inspect binaries. Very useful analysis tools!
  - strip, to strip useless parts of binaries in order to reduce their size
- http://www.gnu.org/software/binutils/
- GPL license

Kernel headers (1)

- The C library and compiled programs needs to interact with the kernel
  - Available system calls and their numbers
  - Constant definitions
  - Data structures, etc.
- Therefore, compiling the C library requires kernel headers, and many applications also require them.
- Available in <linux/...> and <asm/...> and a few other directories corresponding to the ones visible in include/ in the kernel sources

Kernel headers (2)

- System call numbers, in <asm/unistd.h>

#define __NR_exit 1

#define __NR_fork 2

#define __NR_read 3

- Constant denitions, here in <asm-generic/fcntl.h>, included from <asm/fcntl.h>, included from <linux/fcntl.h>

#define O_RDWR 00000002

- Data structures, here in <asm/stat.h>

struct stat {

unsigned long st_dev;

unsigned long st_ino;

[...]

};

Building a toolchain manually

- Building a cross-compiling toolchain by yourself is a difficult and painful task! Can take days or weeks!
  - Lots of details to learn: many components to build, complicated configuration.
  - Lots of decisions to make (such as C library version, ABI, floating point mechanisms, component versions).
  - Need kernel headers and C library sources
  - Need to be familiar with current gcc issues and patches on your platform
  - Useful to be familiar with building and confguring tools
  - See the Crosstool-NG docs/ directory for details on how toolchains are built.

Installing and using a pre-compiled toolchain

- Method 1:
  - Usually, it is simply a matter of extracting a tarball wherever you want.
  - Then, add the path to toolchain binaries in your PATH:
  - Export PATH=/path/to/toolchain/bin/:$PATH
- Method 2:
  - Install package
  - $ sudo apt-get install gcc-arm-linux-gnueabi

SoC and Board Details

Study User Manuals

AM3358 SoC Block Diagram

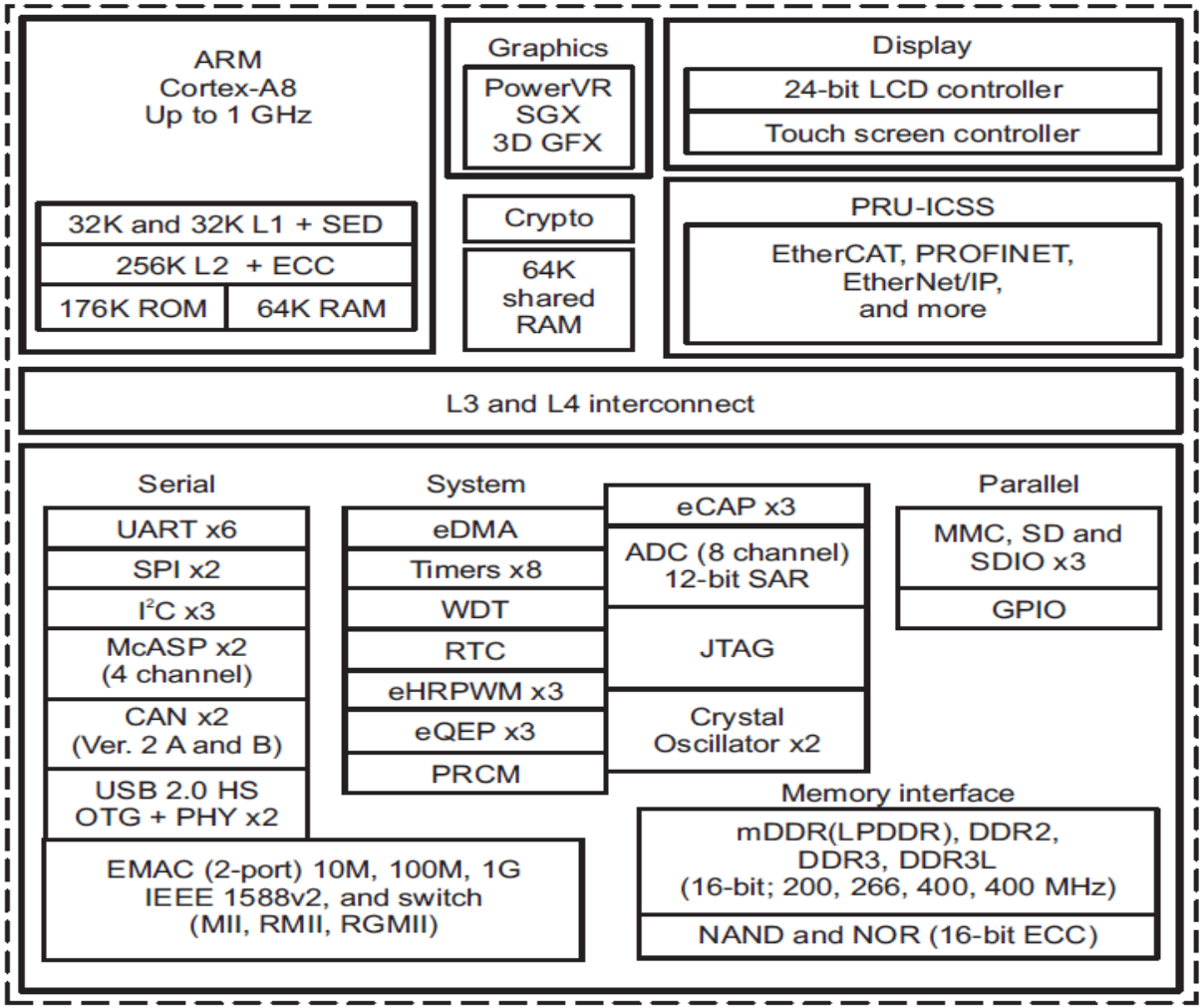
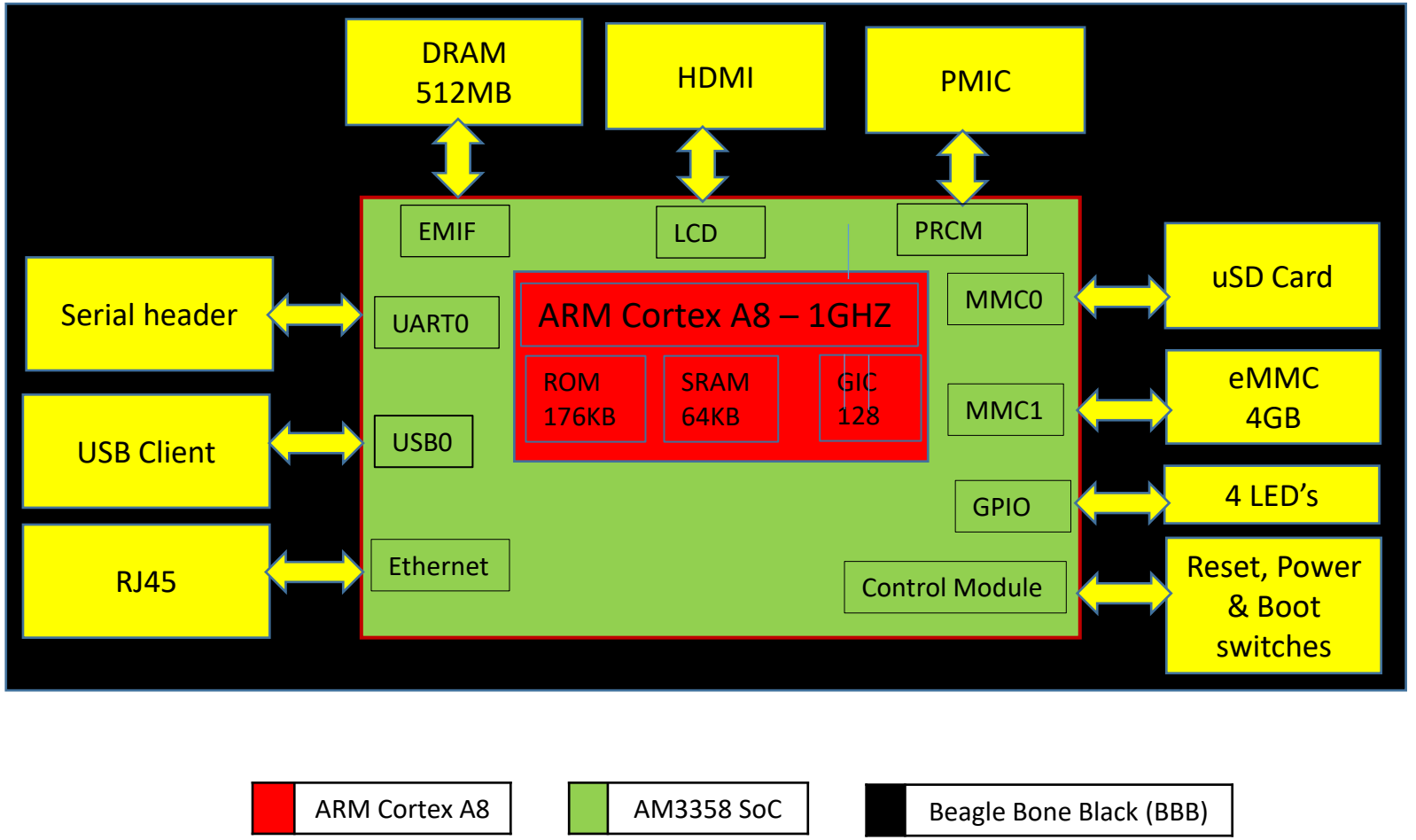
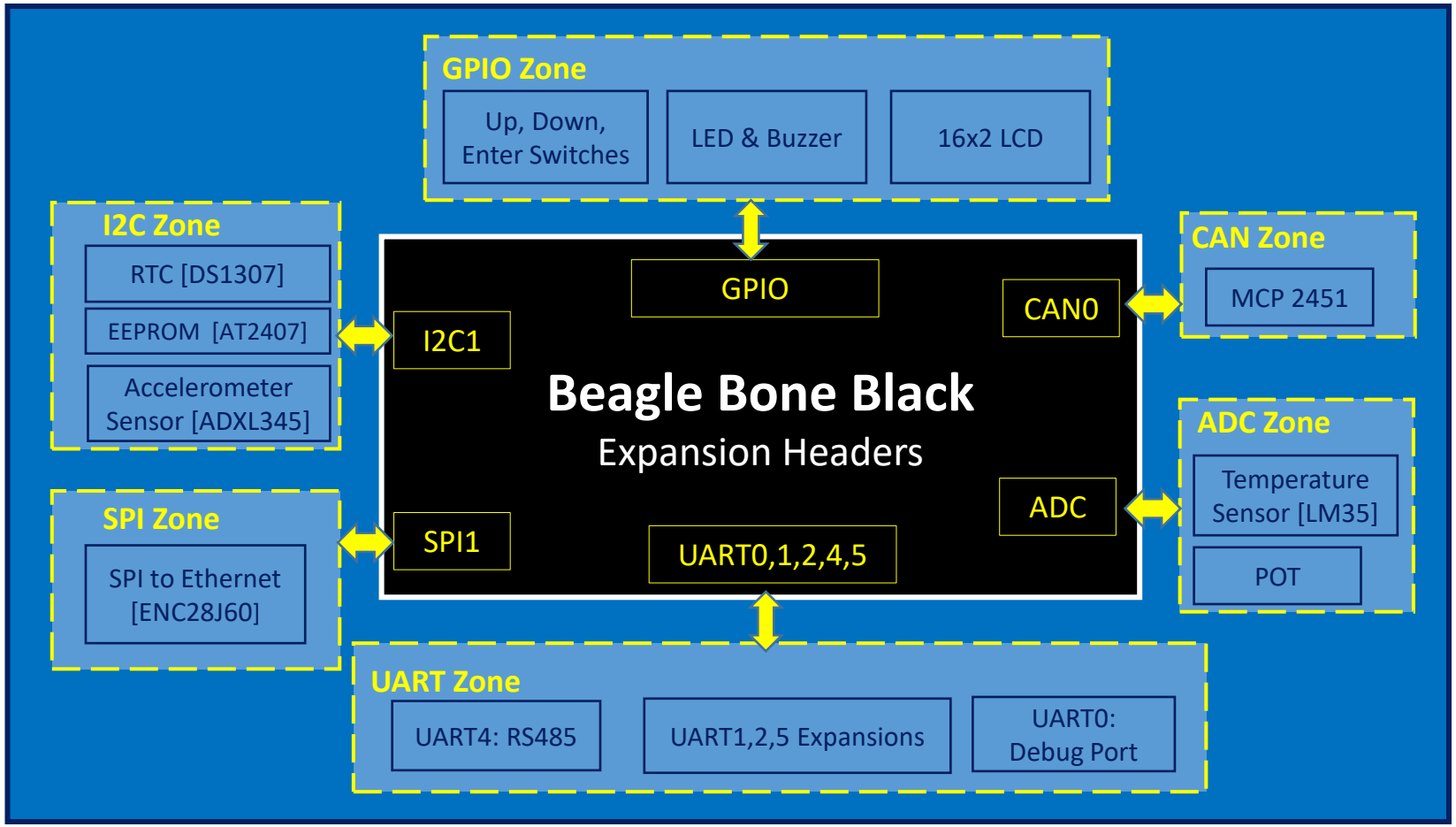


Figure 1-1. AM335x Functional Block Diagram

Beagle Bone Black Block Diagram

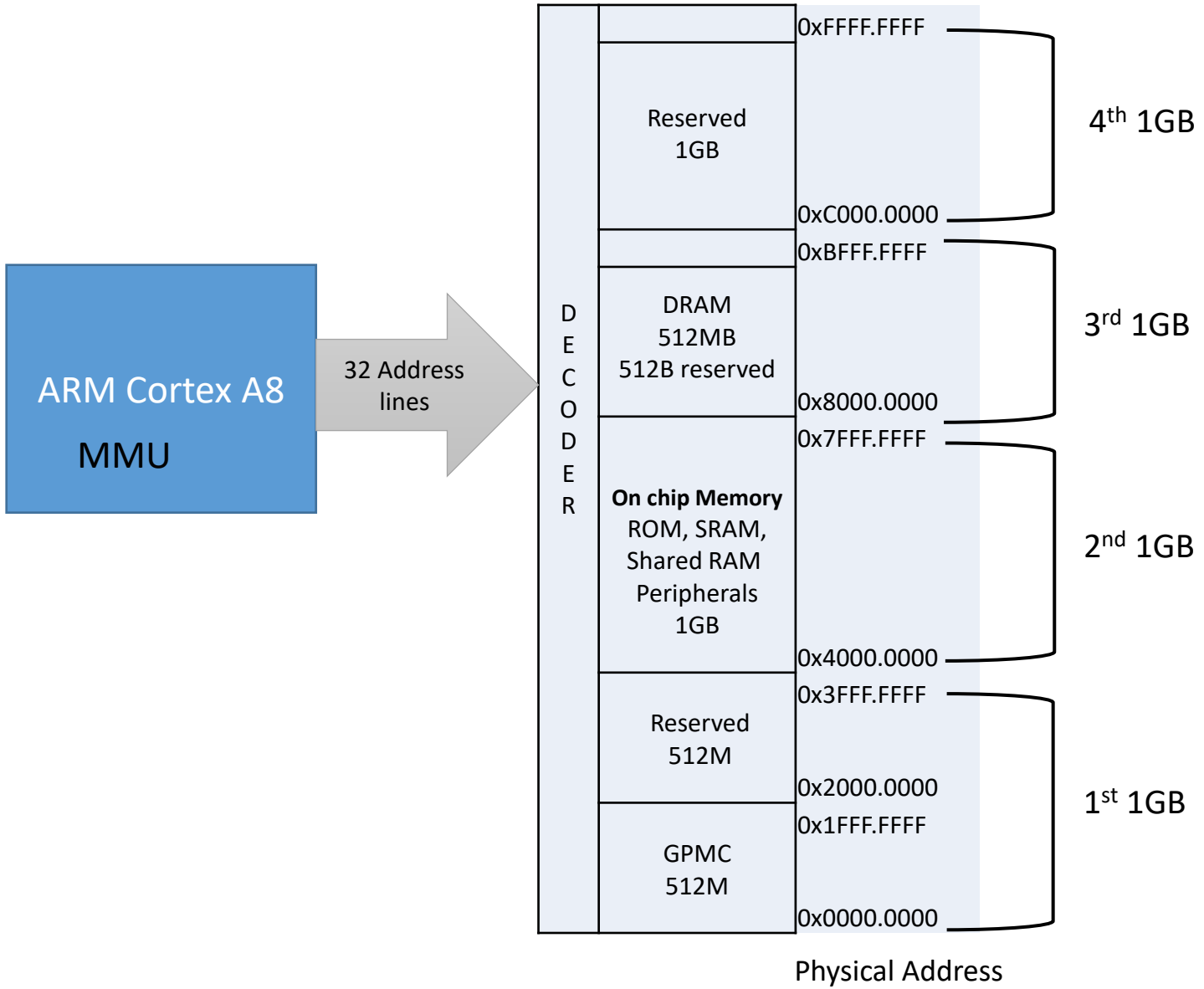


Kernel Masters Beagle Bone Black [KM-BBB] Expansion Board



KERNEL MASTERS

AM335x Memory Mapping (Memory mapped I/O)



Embedded Linux Boot Sequence

Bootloaders

- The bootloader is a piece of code responsible for
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel, from ash storage, from the network, or from another type of non-volatile storage.
  - Possibly decompression of the application binary
  - Execution of the application
- Besides these basic functions, most bootloaders provide a shell with various commands implementing different operations.
- Loading of data from storage or network, memory inspection, hardware diagnostics and testing, etc.

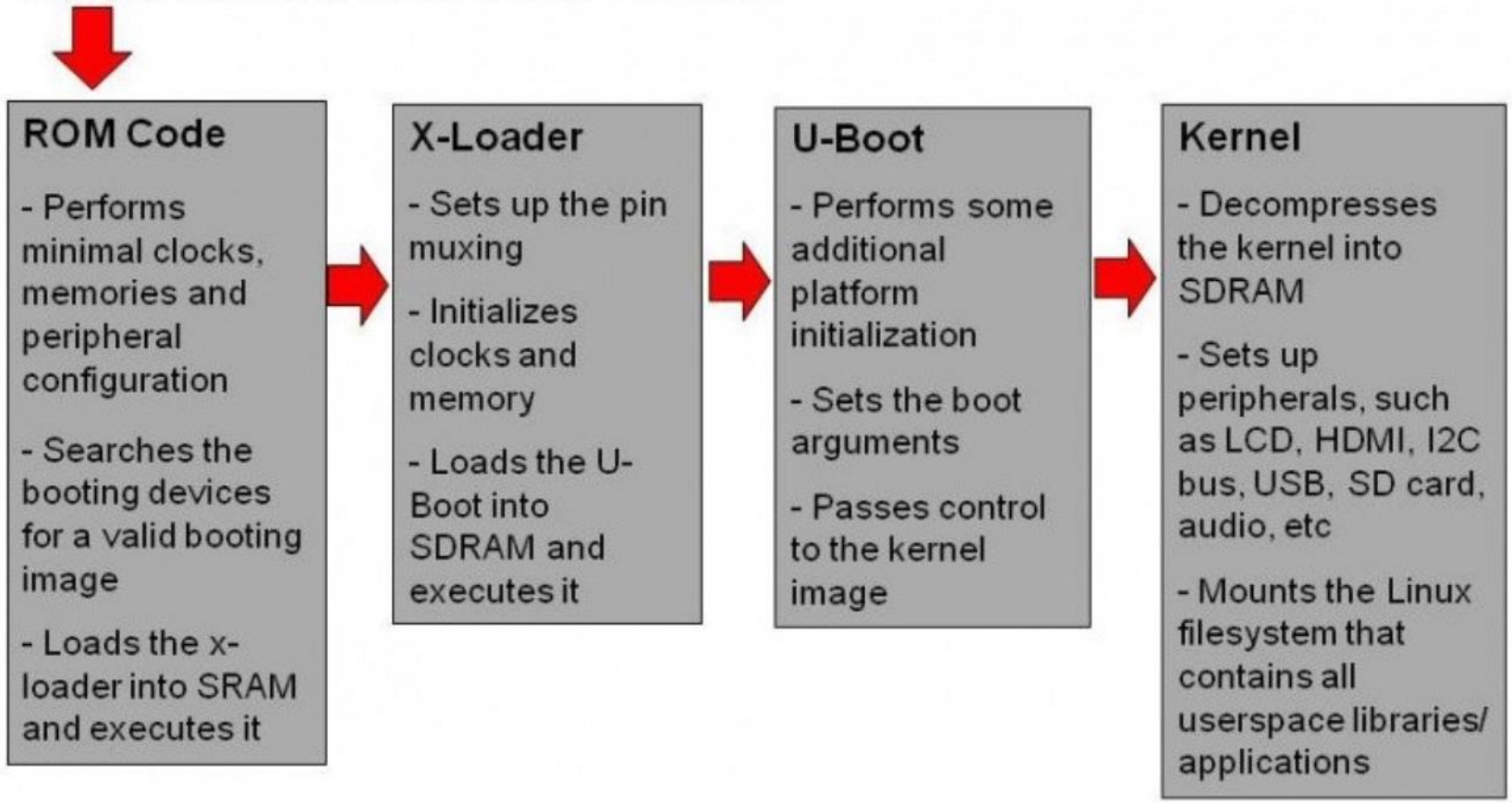
Bootloaders on x86

- The x86 processors are typically bundled on aboard with a non-volatile memory containing a program, the BIOS.
- This program gets executed by the CPU after reset, and is responsible for basic hardware initialization and loading of a small piece of code from non-volatile storage.
- This piece of code is usually the first 512 bytes of a storage device
- This piece of code is usually a 1st stage bootloader, which will load the full bootloader itself.
- The bootloader can then offer all its features. It typically understands filesystem formats so that the kernel file can be loaded directly from a normal filesystem.

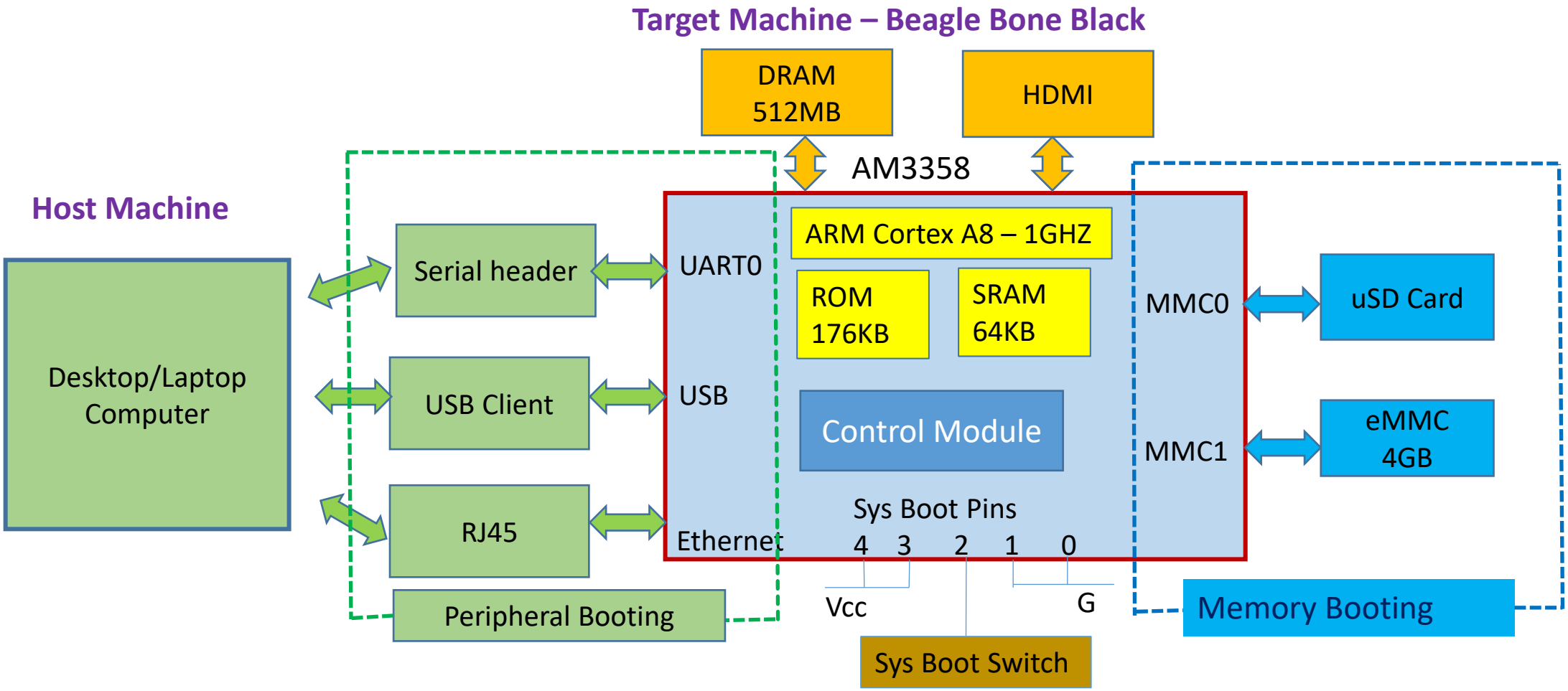
Embedded boot Sequence

- **ROM Code**: tries to and a valid bootstrap image from various storage sources, and load it into SRAM or RAM (RAM can be initialized by ROM code through a conguration header). Size limited to <64 KB. No user interaction possible.
- **X-Loader or U-Boot:** runs from SRAM. Initializes the DRAM, the NAND or MMC controller, and loads the secondary bootloader into RAM and starts it. No user interaction possible. File called MLO.
- **U-Boot:** runs from RAM. Initializes some other hardware devices (network, USB, etc.). Loads the kernel image from storage or network to RAM and starts it. Shell with commands provided. File called u-boot.bin or u-boot.img.
- **Linux Kernel:** runs from RAM. Takes over the system completely (bootloaders no longer exists).

AM3358 boot Sequence



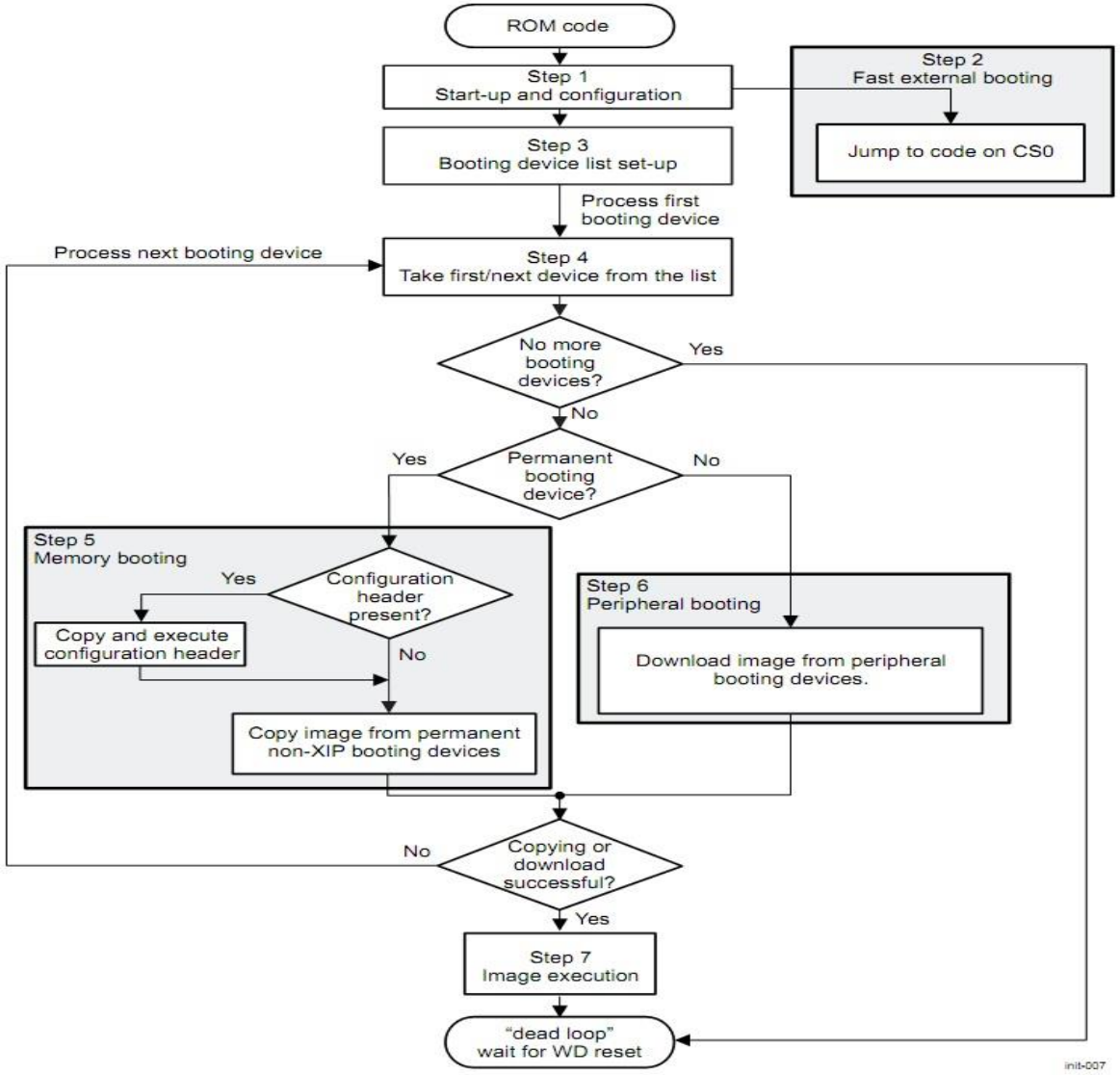
Embedded Linux Boot Sequence



Sysboot SW	Sysboot pins	1st Order	2nd Order	3rd Order	4th Order
OFF	11100	MMC1	MMC0	UART0	USB0
ON	11000	SPIO	MMC0	USB0	UART0

KERNEL MASTERS

AM335x Boot sequence flowchart



Embedded Linux Boot Sequence

For more information about embedded Linux boot sequence click here:

https://kernelmasters.org/blog/2020/06/30/embedded-linux-boot-sequence/

Boot Loader Images

Image Name	Image Size	Image Header	Purpose
spl/u-boot-spl	2.2M	No	The binary of SPL ELF Image.
spl/u-boot-spl.bin	91264 Bytes	No	The second-stage bootloader (a stripped down version of u-boot that fits in SRAM).
spl/u-boot-spl.map	227K	No	contains the information for each symbol.
MLO	91784 Bytes	Yes, GP Header	spl/u-boot-spl.bin with a GP image header prepended to it.
U-boot.bin	474K	No	is the binary compiled U-Boot bootloader
u-boot.map	674K	No	contains the memory map for each symbol
U-boot.img	474K	Yes. GP header	contains u-boot.bin along with an additional header to be used by the boot ROM to determine how and where to load and execute U-Boot.

zImage vs uImage

Image Name	Purpose
Image	The generic Linux kernel binary image file.
zImage	a compressed version of the Linux kernel image that is self-extracting.
uImage	an image file that has a U-Boot wrapper (installed by the mkimage utility) that includes the OS type and loader information.
dtb	Device Tree Blob

⇒ bootz <zImage_addr> - <dtb_addr>
⇒ bootm uImage

zImage vs uImage

Recent versions of U-Boot can boot the zImage binary.

- Older versions require a special kernel image format: uImage
- uImage is generated from zImage using the mkimage tool. It is done automatically by the kernel make uImage target.
- On some ARM platforms, make uImage requires passing a LOADADDR environment variable, which indicates at which physical memory address the kernel will be executed.
- In addition to the kernel image, U-Boot can also pass a Device Tree Blob to the kernel.

The typical boot process is therefore:

1. Load zImage or uImage at address X in memory
2. Load <board>.dtb at address Y in memory
3. Start the kernel with bootz X - Y (zImage case), or bootm X - Y (uImage case)

The - in the middle indicates no initramfs

zImage vs uImage

- A **uImage** includes a U-Boot header which contains the load address that the kernel should be loaded to. This seems nice at first, but then you realize that this makes this entire kernel specific to that load address, defeating the desire to have a kernel that can run "everywhere" (well, at least as broadly as possible).
- A **zImage** does not have this header and therefore the load address must be provided somewhere else. Typically, this is done with a bootz U-Boot command that takes the load address and the Device Tree File address as paramaters:

```
bootz loadaddr - fdtaddr
```

U-Boot Image Format

U-Boot Image Formats:

- U-Boot operates on "image" files which can be basically anything, preceeded by a special header; see thedefinitions in include/image.h for details;
- basically, the header defines the following image properties:
- Target Operating System (Provisions for OpenBSD, NetBSD, FreeBSD, 4.4BSD, Linux, SVR4, Esix,Solaris, Irix, SCO, Dell, NCR, LynxOS, pSOS, QNX, RTEMS, U-Boot, ARTOS, Unity OS, Integrity;
- Currently supported: Linux, NetBSD, VxWorks, QNX, RTEMS, ARTOS, Unity OS, Integrity).
- Target CPU Architecture (Provisions for Alpha, ARM, AVR32, BlackFin, IA64, M68K, Microblaze,MIPS, MIPS64, NIOS, NIOS2, Power Architecture®, IBM S390, SuperH, Sparc, Sparc 64 Bit, Intelx86;
- Currently supported: ARM, AVR32, BlackFin, M68K, Microblaze, MIPS, MIPS64, NIOS,NIOS2, Power Architecture®, SuperH, Sparc, Sparc 64 Bit, Intel x86).•
- Currently supported image compression types: none, gzip, bzip2, lzma, lzo, lz4.
- Load Address
- Entry Point
- Image Name
- Image Timestamp
- The header is marked by a special Magic Number, and both the header and the data portions of the image aresecured against corruption by CRC32 checksums.

Debian 10 porting on BBB

Own built images

Debian 10 prebuilt images porting on KM-BBB

Debian 10 prebuilt images download from the below git repo:

$ git clone git@github.com:kernel-masters/debian10-km-bbb.git

And follow README.md file.

Setup Embedded Linux Development Environment

Own built images

Debian 10 own built images porting on KM-BBB

The below post explain how to porting Debian 10 own built images on Kernel Masters Beagle Bone Black.

https://kernelmasters.org/blog/2020/07/06/setup-embedded-linux-development-environment/

Universal boot loader

@ u-boot level

Universal Boot Loader

- U-boot Overview
- U-boot Source code layout
- U-boot Development Environment
- U-boot Initialization
- U-boot Commands
- U-boot Environment Variables
- U-boot Customization

U-Boot Overview

- U-Boot is a typical free software project
  - License: GPLv2 (same as Linux)
  - Freely available at http://www.denx.de/wiki/U-Boot
  - Documentation available at
    http://www.denx.de/wiki/U-Boot/Documentation
  - The latest development source code is available in a Git repository:
    http://git.denx.de/?p=u-boot.git;a=summary
  - Since the end of 2008, it follows a fixed-interval release schedule. Every two months, a new version is released. Versions are named YYYY.MM.

U-Boot Overview

- The **include/configs/** directory contains one configuration file for each supported board
  - It defines the CPU type, the peripherals and their configuration, the memory mapping, the U-Boot features that should be compiled in, etc.
  - It is a simple .h file that sets C pre-processor constants. See the README file for the documentation of these constants. This file can also be adjusted to add or remove features from U-Boot (commands, etc.).
- Assuming that your board is already supported by U-Boot, there should be one entry corresponding to your board in the boards.cfg file.

Configuring and compiling U-Boot

- **Download u-boot source code from kernel masters GITHUB server.**

  $ cd ~/KM_GITHUB

  $ git clone https://github.com/kernelmasters/beagleboneblack-uboot.git

- **U-boot Configuration**

  $ cd ~/KM_GITHUB/beagleboneblack-uboot

  $ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- am335x_evm_config

- **U-boot Compilation**

  $ cd ~/KM_GITHUB/beagleboneblack-uboot

  $ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-

U-boot prompt

- Connect the target to the host through a serial console.
- Power-up the board. On the serial console, you will see something like:

U-Boot 2013.04 (May 29 2013 - 10:30:21)
OMAP36XX/37XX-GP ES1.2, CPU-OPP2, L3-165MHz, Max CPU Clock 1 Ghz
IGEPv2 + LPDDR/NAND
I2C: ready
DRAM: 512 MiB
NAND: 512 MiB
MMC: OMAP SD/MMC: 0
Die ID #255000029ff800000168580212029011
Net: smc911x-0
KM=>

- The U-Boot shell offers a set of commands. We will study the most important ones, see the documentation for a complete reference or the help command.

U-boot commands

Help information

KM=> help

Version details

KM=> version

U-Boot 2013.04 (May 29 2013 - 10:30:21)

KM Boot Menu

KM=> km_bootmenu

```
#### KM Multi Boot Menu ####

1: Boot From SD Card    [MMC0]
2: Boot From EMMC       [MMC1]
3: Boot From Serial     [UART0]
4: Boot From TFTP       [Default CPSW]
5: Boot With KGDB-KDB   [SDCARD,EMMC,TFTP]
6: Select Ethernet      [CPSW,ENC]
7: Self diagnostic test
0: Stay in Boot Mode

Enter Your Choice:
```

Board Information

KM=> bdinfo

```
arch_number = 0x00000e05
boot_params = 0x80000100
DRAM bank   = 0x00000000
-> start    = 0x80000000
-> size     = 0x20000000
eth0name    = cpsw
ethaddr     = 64:cf:d9:99:5c:50
current eth = cpsw
ip_addr     = 192.168.1.2
baudrate    = 115200 bps
TLB addr    = 0x9fff0000
relocaddr   = 0x9ff56000
reloc off   = 0x1f756000
irq_sp      = 0x9df35ea0
sp start    = 0x9df35e90
Early malloc usage: 410 / 1000
```

Important commands (1)

- The exact set of commands depends on the U-Boot configuration
- help and help command
- **boot**, runs the default boot command, stored in bootcmd
- **bootm <address>** , starts a kernel image loaded at the given address in RAM
- **ext2load**, loads a file from an ext2 filesystem to RAM
  - And also ext2ls to list les, ext2info for information
- **fatload**, loads a file from a FAT filesystem to RAM
  - And also fatls and fatinfo
- **tftp**, loads a file from the network to RAM
- **ping**, to test the network

Important commands (2)

- **loadb, loads, loady,** load a file from the serial line to RAM
- **usb**, to initialize and control the USB subsystem, mainly used for USB storage devices such as USB keys
- **mmc**, to initialize and control the MMC subsystem, used for SD and microSD cards
- **nand**, to erase, read and write contents to NAND flash
- **erase, protect, cp**, to erase, modify protection and write to NOR ash
- **md**, displays memory contents. Can be useful to check the contents loaded in memory, or to look at hardware registers.
- **mm**, modifies memory contents. Can be useful to modify directly hardware registers, for testing purposes.

Environment variables commands

- U-Boot can be configured through environment variables, which affect the behaviour of the different commands.
- Environment variables are loaded from ash to RAM at U-Boot startup, can be modified and saved back to ash for persistence
- There is a dedicated location in ash to store U-Boot environment, defined in the board configuration file.
- Commands to manipulate environment variables:
  - **printenv**, shows all variables
  - **printenv <variable-name>,** shows the value of one variable
  - **setenv <variable-name> <variable-value>,** changes the value of a variable, only in RAM
  - **saveenv**, saves the current state of the environment to flash

Important U-Boot environment variables

- **bootcmd**, contains the command that U-Boot will automatically execute at boot time after a configurable delay, if the process is not interrupted
- **bootargs**, contains the arguments passed to the Linux kernel, covered later
- **serverip**, the IP address of the server that U-Boot will contact for network related commands
- **ipaddr**, the IP address that U-Boot will use
- **netmask**, the network mask to contact the server
- **ethaddr**, the MAC address, can only be set once
- **bootdelay**, the delay in seconds before which U-Boot runs bootcmd
- **autostart**, if yes, U-Boot starts automatically an image that has been loaded into memory

Kernel & DTB images boot from eMMC using u-boot commands

Load kernel image from eMMC first partition boot folder:

KM=> load mmc 1:1 ${loadaddr} /boot/vmlinuz-4.19.94-Kernel-Masters-gc7621ff2c-dirty

Load dtb image from eMMC first partition dtbs folder:

KM=> load mmc 1:1 ${fdtaddr} /boot/dtbs/4.19.94-Kernel-Masters-gc7621ff2c-dirty/km-bbb-am335x.dtb

Setup Kernel command line boot arguments:

KM=> setenv bootargs console=${console} root=/dev/mmcblk1p1 rootfstype=${mmcrootfstype}

Execute kernel image with dtb image:

KM=> bootz ${loadaddr} - ${fdtaddr}

Kernel & DTB images boot from Sdcard using u-boot commands

Load kernel image from sdcard first partition boot folder:

KM=> load mmc 0:1 ${loadaddr} /boot/vmlinuz-4.19.94-Kernel-Masters-gc7621ff2c-dirty

Load dtb image from sdcard first partition dtbs folder:

KM=> load mmc 0:1 ${fdtaddr} /boot/dtbs/4.19.94-Kernel-Masters-gc7621ff2c-dirty/km-bbb-am335x.dtb

Setup Kernel command line boot arguments:

KM=> setenv bootargs console=${console} root=/dev/mmcblk0p1 rootfstype=${mmcrootfstype}

Execute kernel image with dtb image:

KM=> bootz ${loadaddr} - ${fdtaddr}

Scripts in environment variables

- Environment variables can contain small scripts, to execute several commands and test the results of commands.
  - Useful to automate booting or upgrade processes
  - Several commands can be chained using the ; operator
  - Tests can be done using

    if command ; then ... ; else ... ; fi
  - Scripts are executed using run <variable-name>
  - You can reference other variables using ${variable-name}
- **Example**

setenv mmc-boot 'mmc init 0; if fatload mmc 0 80000000 boot.ini; then source; else if fatload mmc 0 80000000 uImage; then run mmcbootargs; bootm; fi; fi'

Script example

```
'if ping 192.8.1.1; then;
                echo "ping success";
else ;
                echo "ping fail";
fi
```

=> setenv test 'if ping 192.8.1.1; then; echo "ping success"; else ; echo "ping fail"; fi'

=> run test

Kernel & DTB images boot from eMMC using script

```
#define KM_UENV_EMMC "setenv bootcmd "\
        "echo board_name=[$board_name] ...;"\
        "echo \"******* Booting From EMMC .... ********\";"\
        "echo board_name=[$board_name] ...;"\
        "setenv fdtfile km-bbb-am335x.dtb;" \
        "setenv console ttyO0,115200n8;"\
        "load mmc 1:1 ${loadaddr} /boot/uEnv.txt;"\
        "env import -t ${loadaddr} ${filesize};" \
        "echo uname_r=[$uname_r] ...;"\
        "echo board_no=[$board_no] ...;"\
        "load mmc 1:1 ${loadaddr} /boot/vmlinuz-${uname_r};"\
        "load mmc 1:1 ${fdtaddr} /boot/dtbs/${uname_r}/km-bbb-am335x.dtb;"\
        "setenv emmc emmc;"\
        "setenv bootargs console=${console} root=/dev/mmcblk1p1 rootfstype=${mmcrootfstype};"\
         "echo ****bootz:Start Kernel****;"\
        "bootz ${loadaddr} ${rdaddr}:${rdsize} ${fdtaddr};"\
        "echo ***END***;"
```

Transferring files to the target

- U-Boot is mostly used to load and boot a kernel image, but it also allows to change the kernel image and the root filesystem stored in flash.
- Files must be exchanged between the target and the development workstation. This is possible:
  - Through the network if the target has an Ethernet connection, and U-Boot contains a driver for the Ethernet chip. This is the fastest and most ecient solution.
  - Through a USB key, if U-Boot support the USB controller of your platform
  - Through a SD or microSD card, if U-Boot supports the MMC controller of your platform.
  - Through the serial port.

What is Board Support Package (BSP)?

- A Board support package (BSP) is essential code for a given computer hardware device that will make that device work with the computers Operating System (OS).
- The BSP Contains a small program called a boot loader or boot manager the places the OS and device drivers into memory.
- The Content of the BSP depend on the particular hardware and OS.
- Specific tasks that the BSP performs include the following, in order:
  - Initialize the processor.
  - Initialize the bus.
  - Initialize the Interrupt Controller.
  - Initialize the clock.
  - Initialize the RAM settings.
  - Run the bootloader.
- In addition, a BSP can contain directives, compilation parameters, and hardware parameters for configuring the OS.

U-boot Source code Layout

/arch	Architecture specific files
/arc	Files generic to ARC architecture
/arm	Files generic to ARM architecture
/m68k	Files generic to m68k architecture
/microblaze	Files generic to microblaze architecture
/mips	Files generic to MIPS architecture
/nds32	Files generic to NDS32 architecture
/nios2	Files generic to Altera NIOS2 architecture
/openrisc	Files generic to OpenRISC architecture
/powerpc	Files generic to PowerPC architecture
/riscv	Files generic to RISC-V architecture
/sandbox	Files generic to HW-independent "sandbox"
/sh	Files generic to SH architecture
/x86	Files generic to x86 architecture
/api	Machine/arch independent API for external apps
/board	Board dependent files

/cmd	U-Boot commands functions
/common	Misc architecture independent functions
/configs	Board default configuration files
/disk	Code for disk drive partition handling
/doc	Documentation (don't expect too much)
/drivers	Commonly used device drivers
/dts	Contains Makefile for building internal U-Boot fdt.
/examples	Example code for standalone applications, etc.
/fs	Filesystem code (cramfs, ext2, jffs2, etc.)
/include	Header Files
/lib	Library routines generic to all architectures
/Licenses	Various license files
/net	Networking code
/post	Power On Self Test
/scripts	Various build scripts and Makefiles
/test	Various unit test files
/tools	Tools to build S-Record or U-Boot images, etc.

X-loader (SPL) Initialization Flow

Processor Specific Initialization:

arch/arm/cpu/armv7/start.S:

reset: -> cpu_init_cp15 -> cpu_init_crit -> lowlevel_init

arch/arm/cpu/armv7/lowlevel_init.S: (setup pll,mux,memory)

lowlevel_init ->

For boards with SPL this should be empty since SPL can do all of this init in the SPL **board_init_f()** function which is called immediately after this.

bl s_init

SoC Specific Initialization:

arch/arm/mach-omap2/am33xx/board.c: [PRCM]

board_init_f: -> hw_data_init() -> early_system_init() -> sdram_init();

board_early_init_f() ->

prcm_init() -> set_mux_conf_regs()

Board Specific Initialization:

board/ti/am335xx/board.c:

- set_mux_conf_regs() -> enable_board_pin_mux()

board/ti/am335x/mux.c:

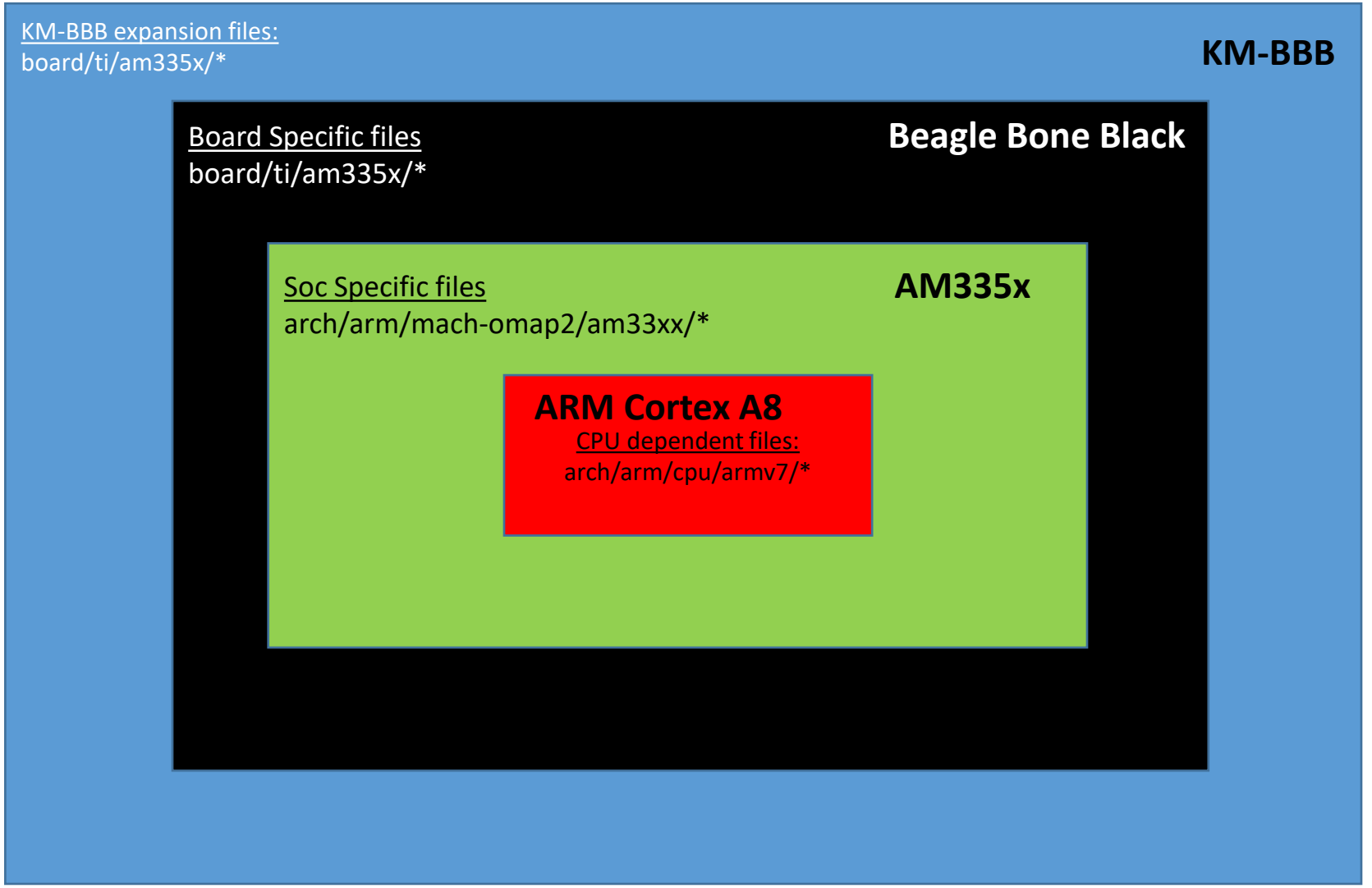
U-Boot Initialization Flow

common/main.c:
Main_loop() -> autoboot_command()

common/autoboot.c:
Autoboot_command() -> multiboot() -> lcd_bootmenu()

board/ti/am335x/km_lcd.c:
lcd_bootmenu() ->

U-boot Source code Initialization



Communicate with hardware

@ u-boot level

Communicate with Hardware (Memory Mapped I/O)

Read CONTROL STATUS Register:

Control Module Base Address is 0x44E1.0000 ; CONTROL STATUS Register offset address is 0x40

PA of CONTROL STATUS REG = Base Address of control module + offset address of control status register

```
= 0x44E10000
+          40
----------------------
  0x 44E10040
----------------------------
```

Physical Address = Base Address + Offset Address

Read control status register using md command:

=> md <address> [offset]

⇒ md 0x44E10040 (read word of data)

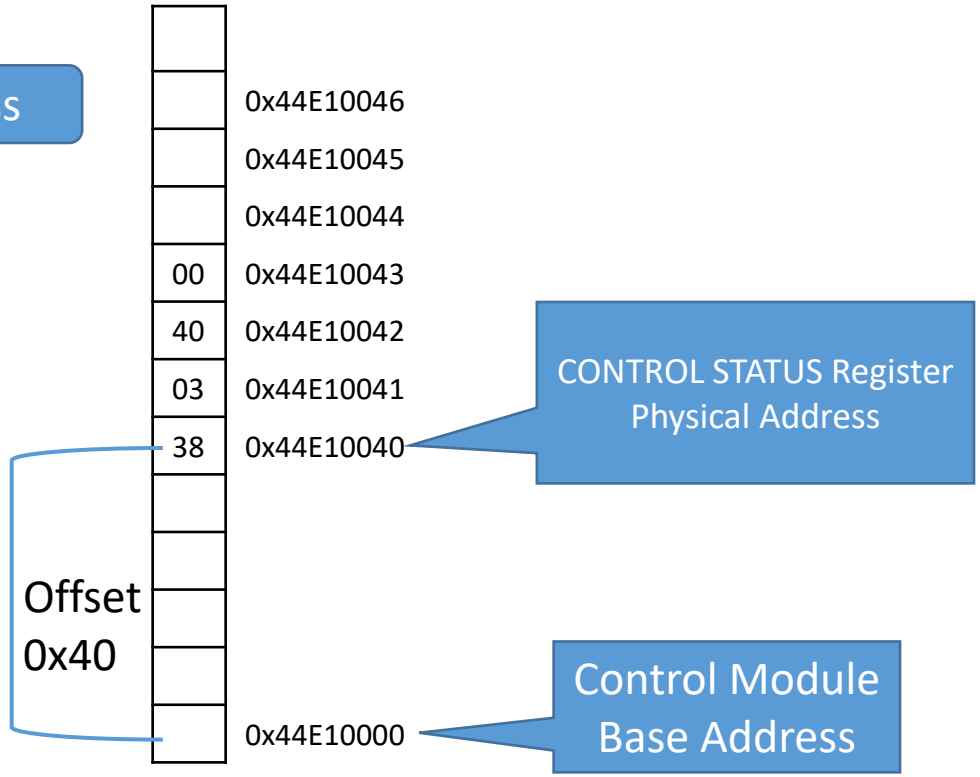
⇒ md.b 0x44E10040 (read byte of data)

=> md 0x44E10040

44e10040: 00400338 00000000 00000000 00000000

SYS Boot Switch OFF – 11**1**00

ON – 11**0**00



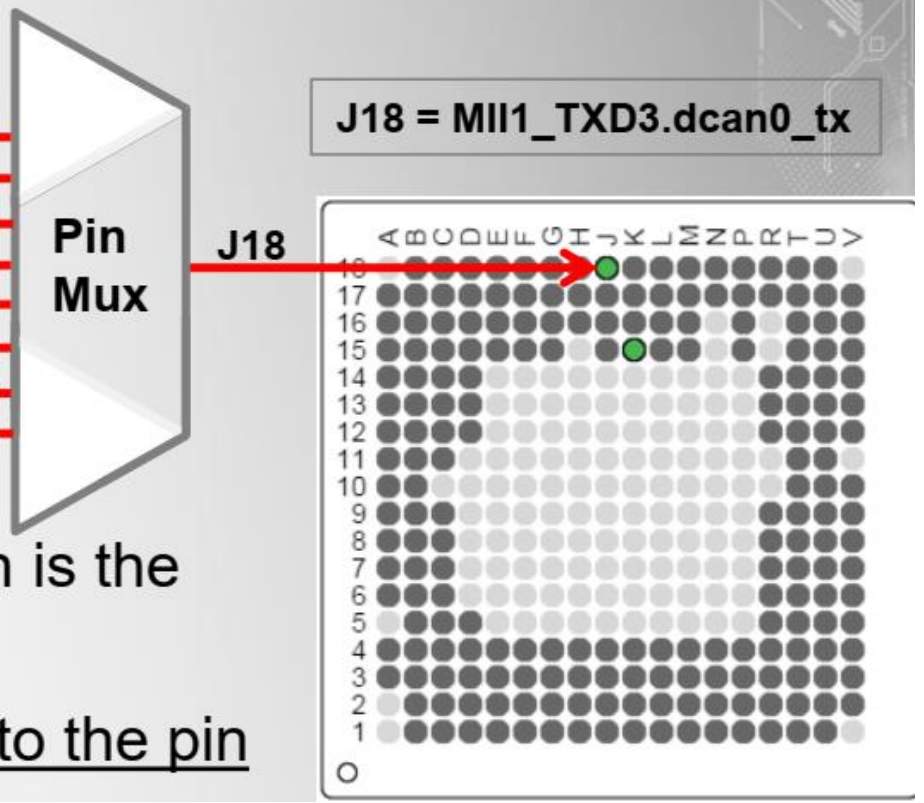
Mux/Pad Configuration

@ u-boot level

For SOC that use pin muxes - what is a pin mux?

- Most pins on the SOC have a mux must be set to enable a peripheral access
- Each pin name has several signal names that can accessed by a mux mode

ZCE BALL NUMBER [1]	ZCZ BALL NUMBER [1]	PIN NAME [2]	SIGNAL NAME [3]	MODE [4]	TYPE [5]
M17	J18	MII1_TXD3	gmii1_txd3	0	O
			dcan0_tx	1	O
			rgmii1_td3	2	O
			uart4_rxd	3	I
			mcasp1_fsx	4	I/O
			mmc2_dat1	5	I/O
			mcasp0_fsr	6	I/O
			gpio0_16	7	I/O



- In this example J18 is using mux mode 1 which is the DCAN0 TX signal
- The silicon IP in the DTS has to be connected to the pin

Pad Control Registers: TRM – page no: 1369

9.3 Registers

9.3.1 CONTROL_MODULE Registers

Table 9-10 lists the memory-mapped registers for the CONTROL_MODULE. All other register offset addresses not listed in Table 9-10 should be considered as reserved locations and the register contents should not be modified.

Table 9-10. CONTROL_MODULE REGISTERS

Offset	Acronym	Register Description	Section
0h	control_revision		Section 9.3.1.1
4h	control_hwinfo		Section 9.3.1.2
10h	control_sysconfig		Section 9.3.1.3
40h	control_status		Section 9.3.1.4
110h	control_emif_sdram_config		control_emif_sdram_config Register (offset = 110h) [reset = 0h]
428h	core_sldo_ctrl		Section 9.3.1.5
42Ch	mpu_sldo_ctrl		Section 9.3.1.6
444h	clk32kdivratio_ctrl		Section 9.3.1.7
448h	bandgap_ctrl		Section 9.3.1.8
44Ch	bandgap_trim		Section 9.3.1.9
458h	pll_clkinpulow_ctrl		Section 9.3.1.10
468h	mosc_ctrl		Section 9.3.1.11
470h	deepsleep_ctrl		Section 9.3.1.12

GPIO Registers: TRM – Page no: 4881

25.4 GPIO Registers

25.4.1 GPIO Registers

Table 25-5 lists the memory-mapped registers for the GPIO. All register offset addresses not listed in Table 25-5 should be considered as reserved locations and the register contents should not be modified.

Table 25-5. GPIO Registers

Offset	Acronym	Register Name	Section
0h	GPIO_REVISION		Section 25.4.1.1
10h	GPIO_SYSCONFIG		Section 25.4.1.2
20h	GPIO_EOI		Section 25.4.1.3
24h	GPIO_IRQSTATUS_RAW_0		Section 25.4.1.4
28h	GPIO_IRQSTATUS_RAW_1		Section 25.4.1.5
2Ch	GPIO_IRQSTATUS_0		Section 25.4.1.6
30h	GPIO_IRQSTATUS_1		Section 25.4.1.7
34h	GPIO_IRQSTATUS_SET_0		Section 25.4.1.8
38h	GPIO_IRQSTATUS_SET_1		Section 25.4.1.9
3Ch	GPIO_IRQSTATUS_CLR_0		Section 25.4.1.10
40h	GPIO_IRQSTATUS_CLR_1		Section 25.4.1.11
44h	GPIO_IRQWAKEN_0		Section 25.4.1.12
48h	GPIO_IRQWAKEN_1		Section 25.4.1.13
114h	GPIO_SYSSTATUS		Section 25.4.1.14
130h	GPIO_CTRL		Section 25.4.1.15
134h	GPIO_OE		Section 25.4.1.16
138h	GPIO_DATAIN		Section 25.4.1.17
13Ch	GPIO_DATAOUT		Section 25.4.1.18
140h	GPIO_LEVELDETECT0		Section 25.4.1.19

MuX/Pad Configuration

What is Mux Configuration:

Most of the pins in microcontroller consists of more than one functionality.

Using multiplexer to select the required functionality. That is called Mux/pad configuration

LCD_DATA14 Mux Register

MUX MODE

PA of LCD_DATA14 pad Reg = CM base add + offset

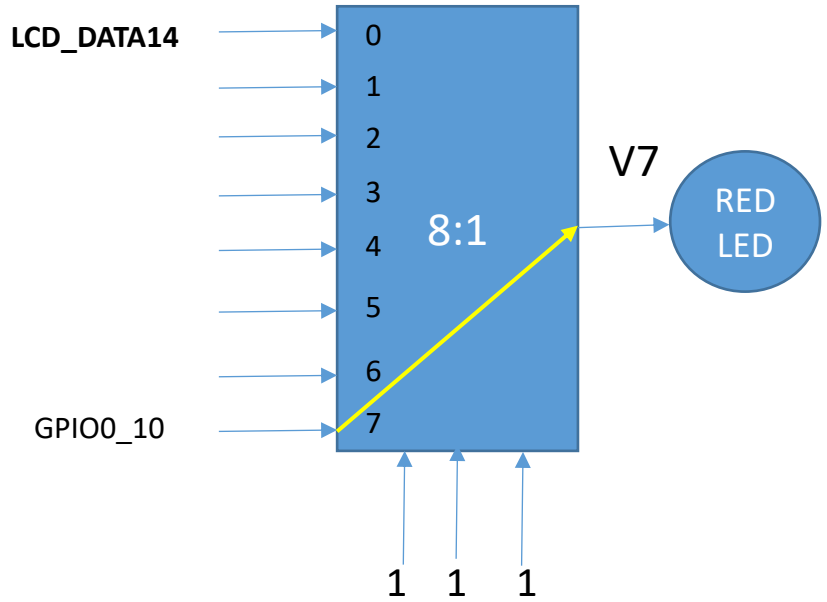
= 0x44E10000

\+ 8D8

0x44E108D8 (PA of LCD_DATA14 MUX)

PA of GPIO_OE Register = 0x44E07000 + 0x134 = **0x44E07134**

PA of GPIO_DATAOUT Reg = 0x44E07000 + 0x13C = **0x44E0713C**



Control LED Device from u-boot prompt

To Control LED Device two things required:

1. LED Configuration (Activate):

   Load mode bits 7 in LCD_DATA14 Pad register using mm command to **enable GPIO MUX**.

   Clear 10th pin in GPIO_OE register using mm command to **set output direction**.

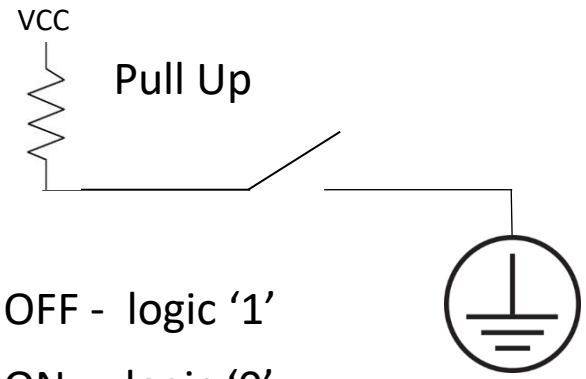
2. LED Operation:

   Set 10th bit in GPIO_DATA register using mm command to LED ON.

   Clear 10th bit in GPIO_DATA register using mm command to LED OFF.

Types of switches

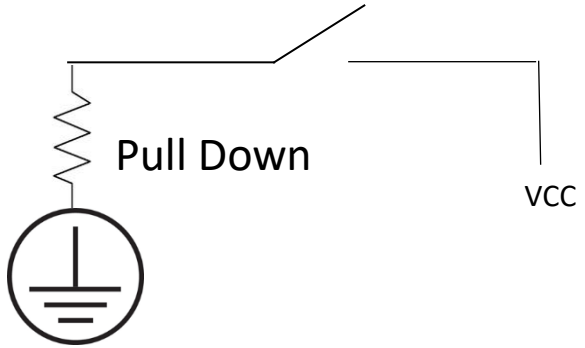
-ve level Switch



OFF - logic ‘1’

ON - logic ‘0’

+ve level Switch



OFF - logic ‘0’

ON - logic ‘1’

Pull up: Pin connected to supply
Pull down: pin connected to ground

GPIO0_11 configured as an either –ve level or + ve level switches
GPIO0_26 & GPIO0_27 configured as –ve level switches

Get switch value from u-boot prompt

To get switch value two things required:

1. Switch Configuration (Activate):

   Load mode 3 bits in LCD_DATA15 Pad register using mm command to **enable GPIO MUX**.

   Clear 3rd,4th bit in LCD_DATA15 Pad register to configured as PULL Down

   Set 5th bit to configured as input.

   Set 11th bit in GPIO_OE register using mm command to **set input direction**. (By default input direction).

2. Get switch value:

   Read 11th bit in GPIO_DATAIN register using md command to get switch value..

Get switch value from u-boot prompt

PA of LCD_DATA15 pad Reg = CM base add + offset
= 0x44E10000
+ 8DC

0x44E108DC (PA of LCD_DATA15 MUX)

PA of GPIO_OE Register = 0x44E07000 + 0x134 = **0x44E07134**
PA of GPIO_DATAIN Reg = 0x44E07000 + 0x138 = **0x44E07138**

0x40000000

Pull Down: 0x27

ON: 0x0040 -> 0000 0000 0100 0000

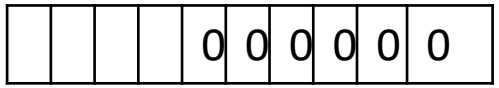
OFF: 0x0840 -> 0000 1000 0100 0000

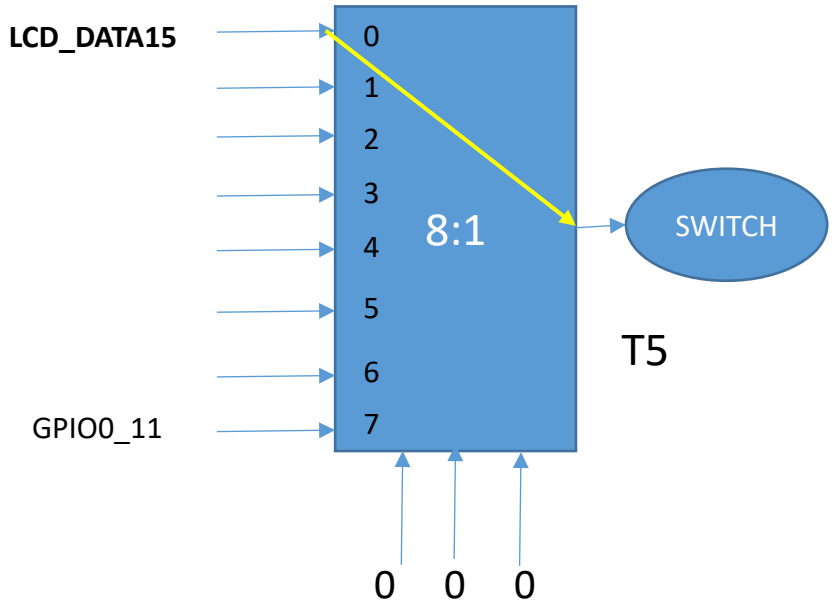
Pull Up: 0x37

OFF: 0x0040 -> 0000 0000 0100 0000

ON: 0x0840 -> 0000 1000 0100 0000

LCD_DATA14 Mux Register





Read/write hardware registers

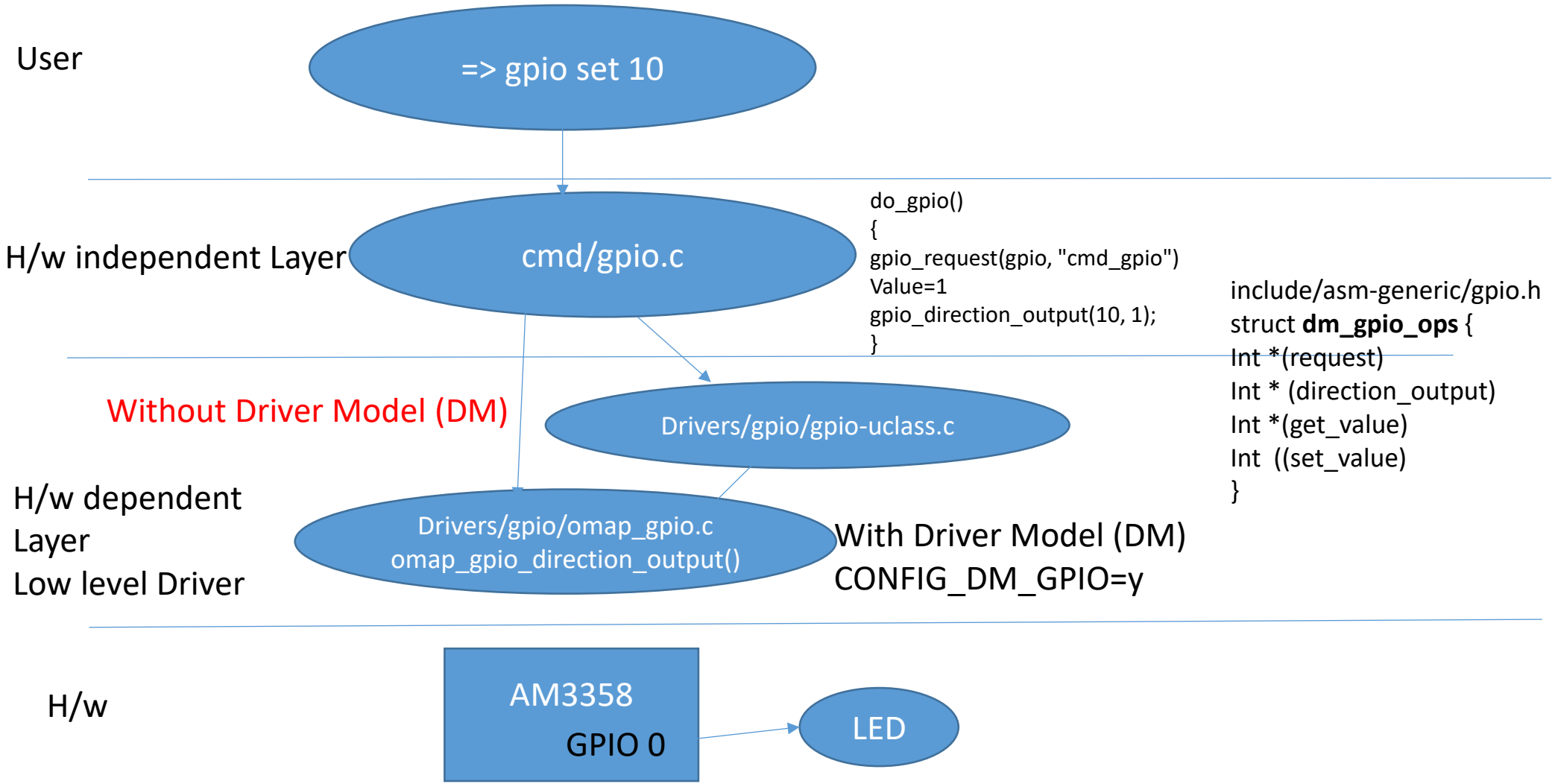
arch/arm/include/asm/arch-am33xx/mux_am33xx.h

__raw_writel()

```
#define __raw_writel(v,a)       __arch_putl(v,a)
#define __arch_putl(v,a)            (*(volatile unsigned int *)(a) = (v))
```

arch/arm/include/asm/io.h

GPIO Framework in U-boot



What is Device Tree?

How uboot pass the hardware information to the kernel?

Approach 1: ATAGS (Before the Device Tree):

The kernel contains the entire description of the hardware.

The bootloader loads a single binary, the kernel image, and executes it. uImage or zImage.

a) ATAGS interface. Minimal information is passed from firmware to the kernel with a tagged list of predefined parameters.

r0 : 0

r1 : Machine type number

r2 : Physical address of tagged list in system RAM

U-Boot command: bootm <kernel img addr>

Approach 2: (Booting with Device Tree)

The kernel no longer contains the description of the hardware, it is located in a separate binary: the **device tree blob (dtb)**.

The bootloader loads two binaries: the **kernel image** and the **DTB**.

Kernel image remains **uImage** or **zImage**

Approach 2: (Booting with Device Tree)

Device Tree (DT), is a data structure and language for describing hardware.

More specifically, it is a description of hardware that is readable by an operating system so that the operating system doesn't need to hard code details of the machine.

DTB located in arch/arm/boot/dts, one per board

Approach 2: (Booting with Device Tree)

r0 : 0

r1 : Valid machine type number. When using a device tree, a single machine type number will often be assigned to represent a class or family of SoCs.

http://www.arm.linux.org.uk/developer/machines/

r2 : physical pointer to the device-tree block in RAM.

Device tree can be located anywhere in system RAM, but it should be aligned on a 64 bit boundary.

U-Boot command:

boot[mz] <kernel img addr> - <dtb addr>