

Pr. Didier Donsez (didier.donsez@imag.fr)

Pr. Olivier Gruber (olivier.gruber@imag.fr)

Laboratoire d'Informatique de Grenoble
Université de Grenoble-Alpes

Two Parts

- About individual machines – Pr. Olivier Gruber – 50%
 - About sensors, embedded systems, gateways
 - Step1 – Introduction to bare-metal coding and core tools
 - Step2 – Your own minimal Linux distribution
- About distributed architectures – Pr. Didier Donsez – 50%
 - Infrastructure to gather and process information

First Part – Overview

- About individual machines
 - Inverted pedagogy --- **Team learning, individual work**
 - A work log to read, understand, and complete
 - The work log is for yourself first
 - No exam
 - You will surrender your Work Log and your code
 - You need to keep me informed of your progress along the path...



Using which tools?



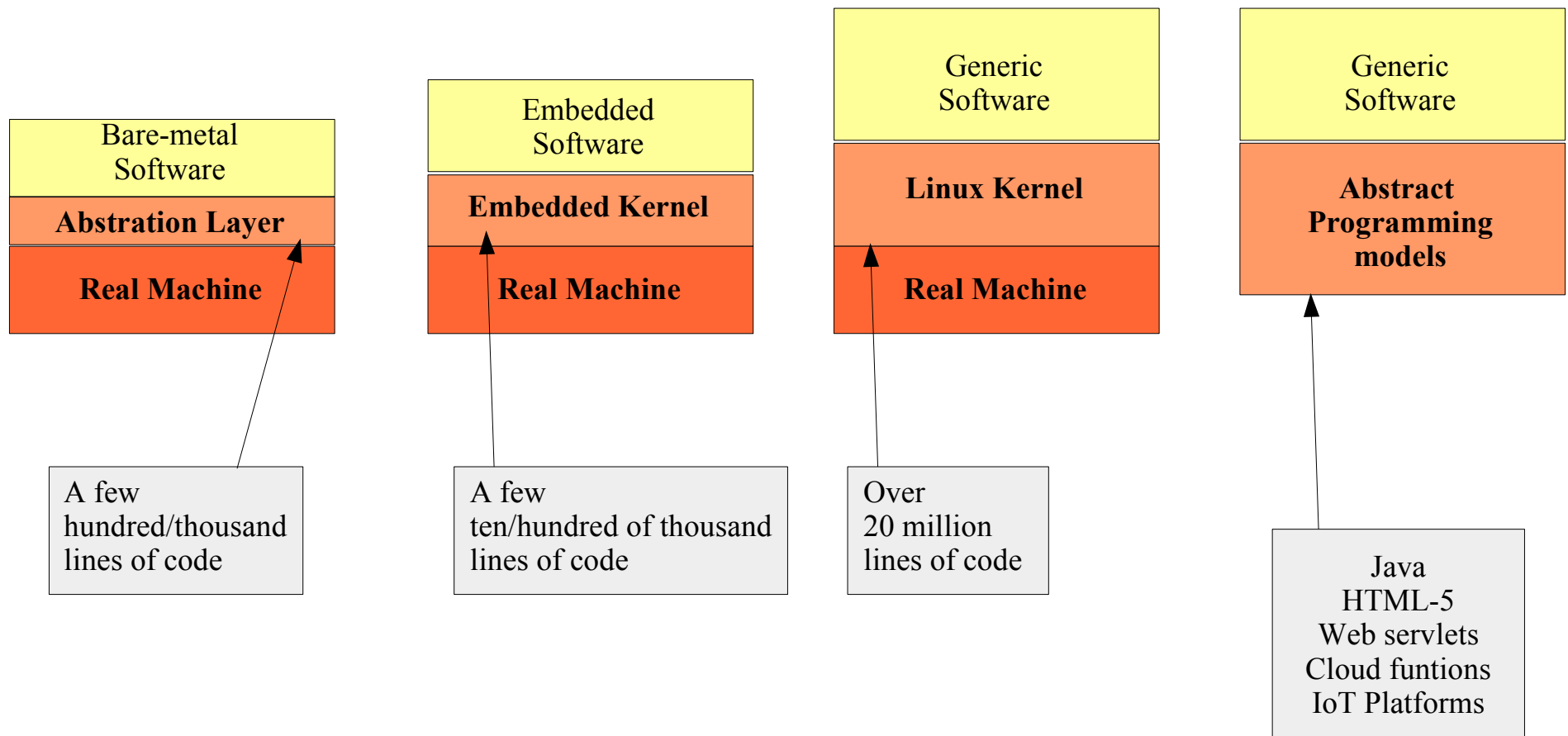
How do we program these?

Methodology

- Work log as a guide...
 - The destination is not the goal
 - The goal is the path, the learning along that path
 - You need to be inquisitive and independent
- Learning a know-how
 - Shell programming and a few core system tools
 - C programming and a bit of assembly programming
 - Debugging with gdb
 - Using Qemu for bare-metal programming
 - The boot process explained
 - Configuring and building your own Linux kernel
 - Special file systems (/dev and /proc)
 - Minimal distribution

Background – Perspective

- Very diverse programming environments



Background – Perspective

- **Linux Kernel vs Distribution**

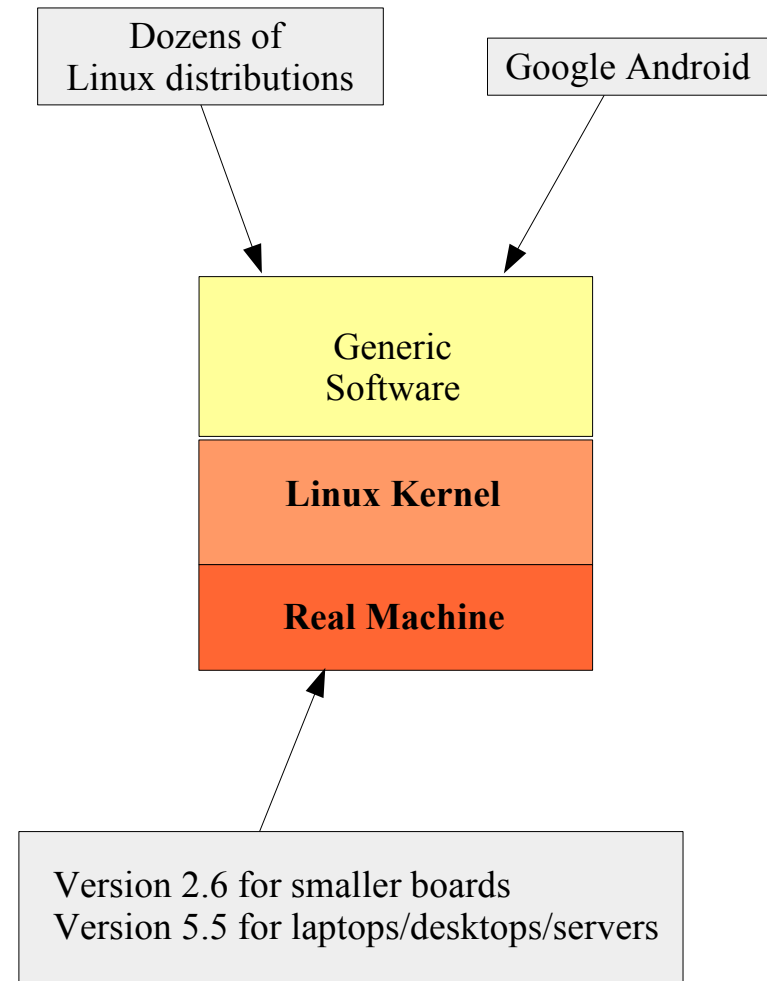
- Linux kernel → files and processes
- Distribution → services and tools

- **Google Android**

- Linux kernel
- A few C libraries (webkit, codecs, etc.)
- Java environment for services and tools

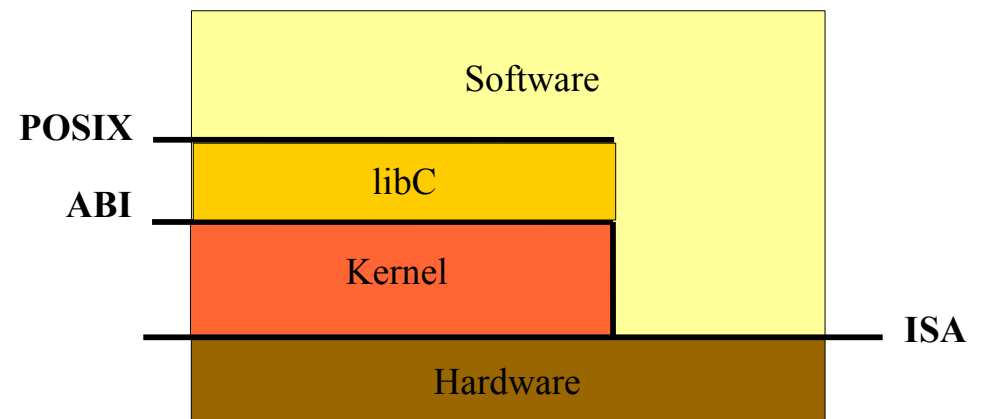
- **Kernel versions**

- Many versions
- Each distribution requires a specific version
- Sometimes, a patched version



Background – Perspective

- Each Linux distribution – each a different special mix
 - Instruction Set Architecture (ISA) – processor specific
 - Defines the instruction set
 - Defines other concepts such as page tables, traps, interrupts, etc.
 - Application Binary Interface (ABI)
 - Defines the kernel interface with 337 syscalls⁽¹⁾
 - Often exposed as a Virtual Dynamic Shared Object (VDSO)
 - Shared core libraries
 - Like the POSIX⁽²⁾ libC from the GNU⁽³⁾ toolchain



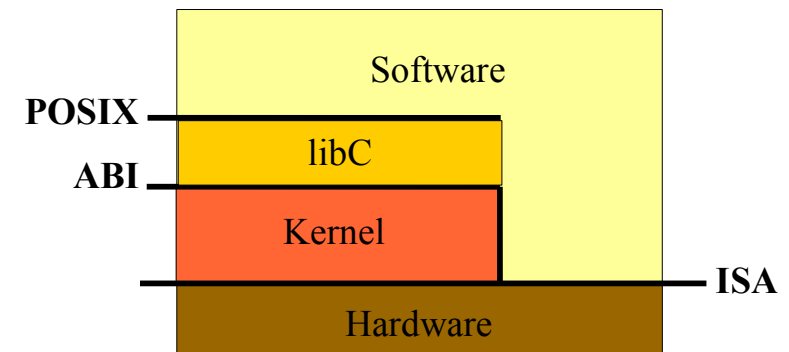
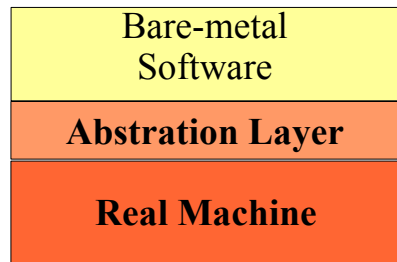
(1) <https://syscalls.kernelgrok.com>

(2) Portable Operating System Interface (IEEE standards)

(3) <http://www.gnu.org>

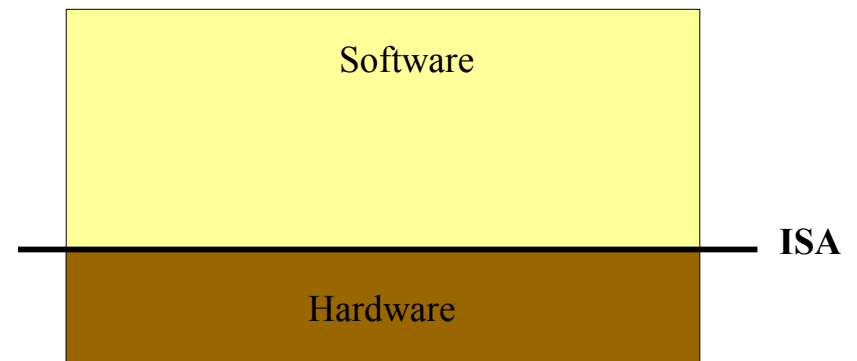
Course Content

- Bare-metal development
 - Getting started with software that runs directly on the "*bare metal*"
- Minimal Linux distribution
 - Customize the Linux kernel
 - Create a minimal root file system
 - Master our own mix...

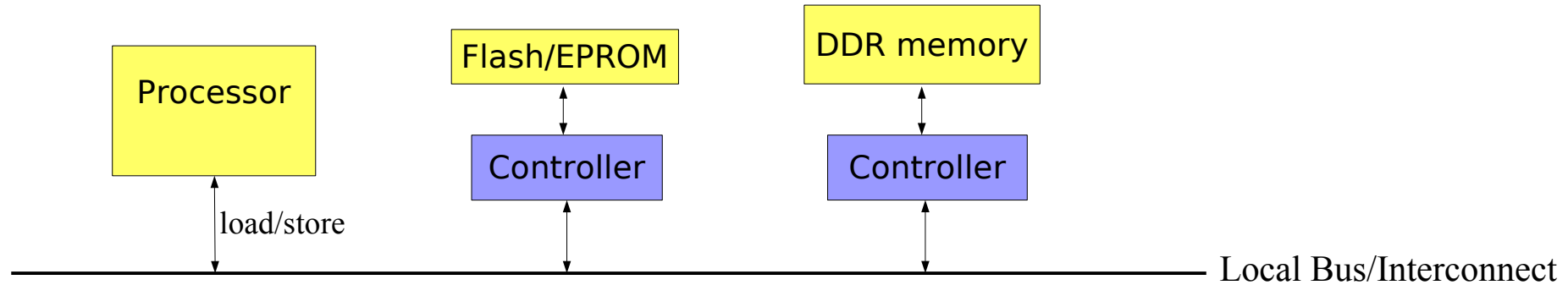


Bare-metal Software

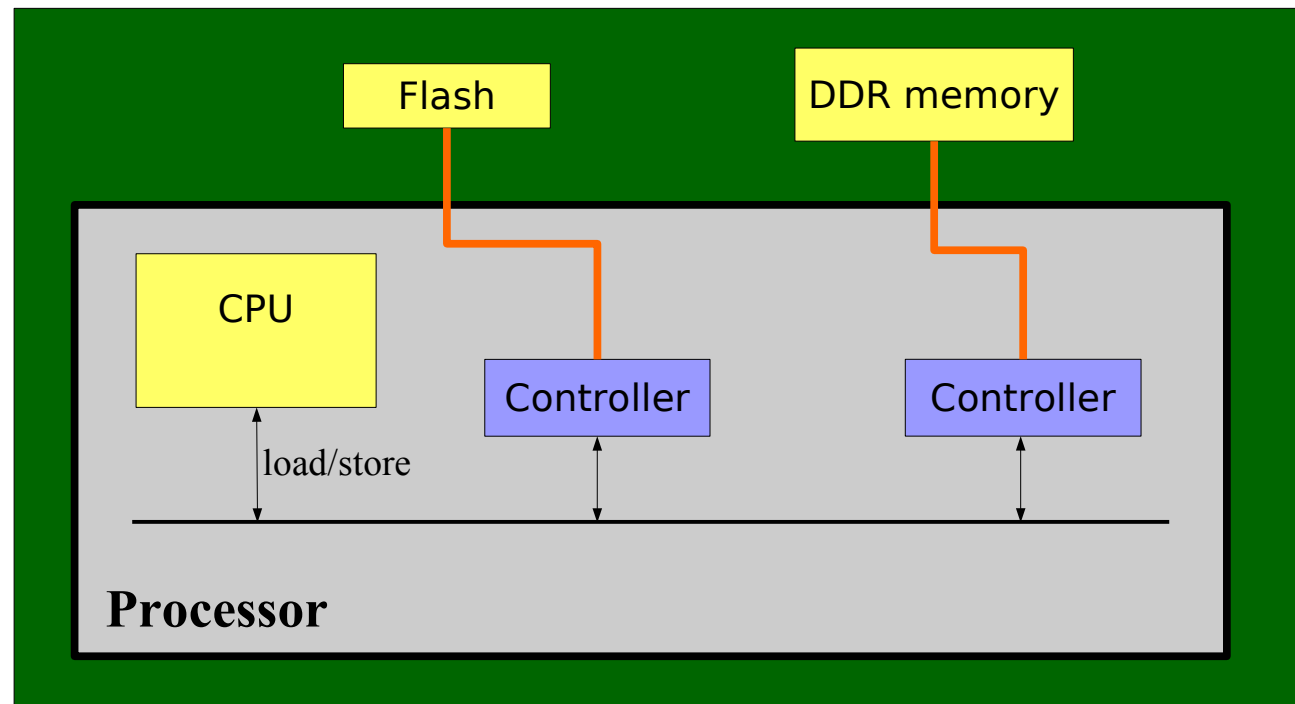
- Bare-metal Software
 - Runs directly on the "*bare metal*"
- Instruction Set Architecture (ISA)
 - Defines the instruction set
 - Defines other concepts such as page tables, traps, interrupts, etc.



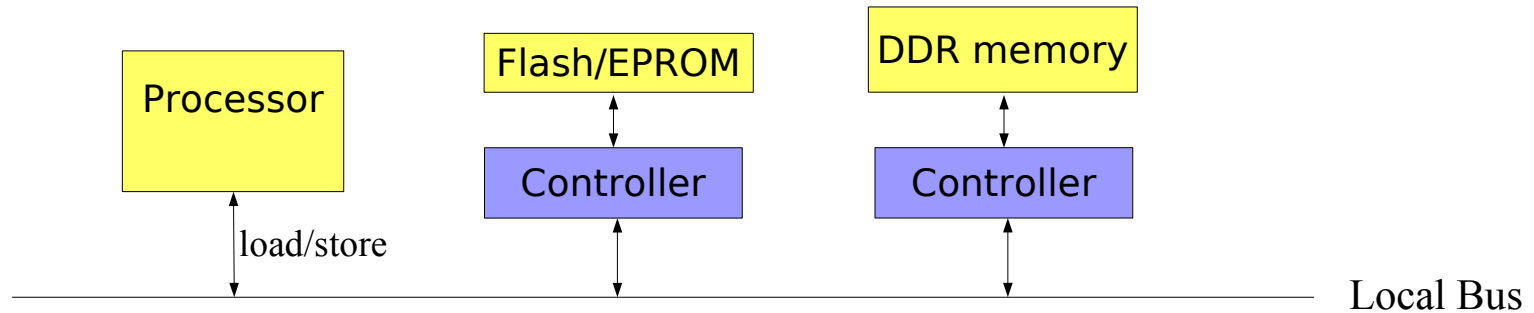
Hardware – Basics



Board

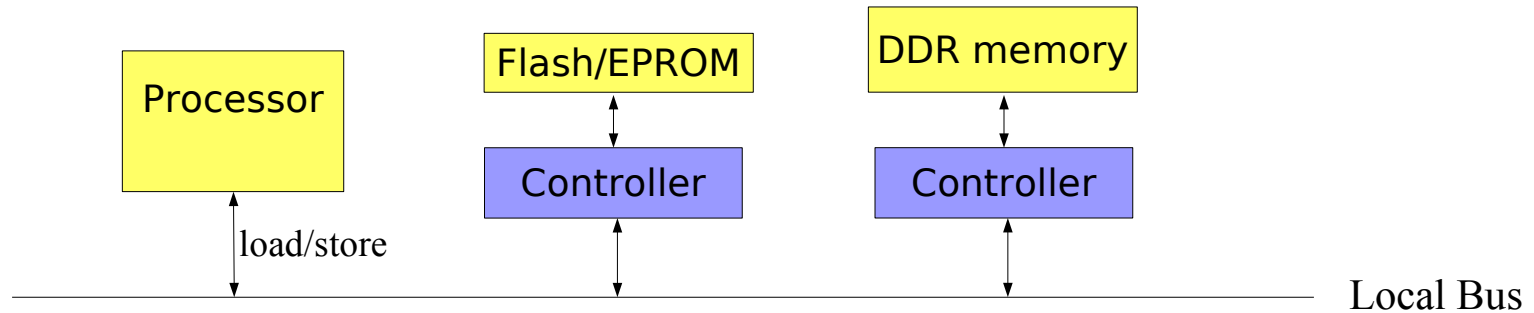


Hardware – Reset / Power-up



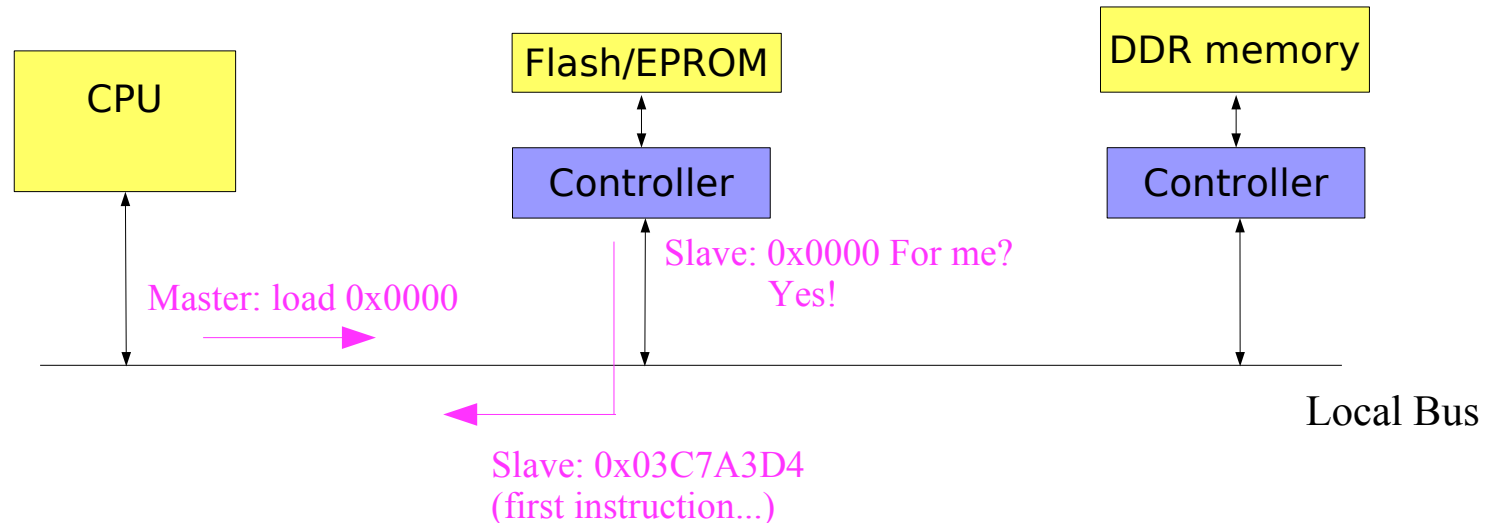
- **Processor**
 - Executes instructions internally
 - **Only knows how to issue load/store operations on the bus**
- **Bus**
 - Data wires + control wires
 - Routes load/store operations to controllers

Hardware – Reset / Power-up



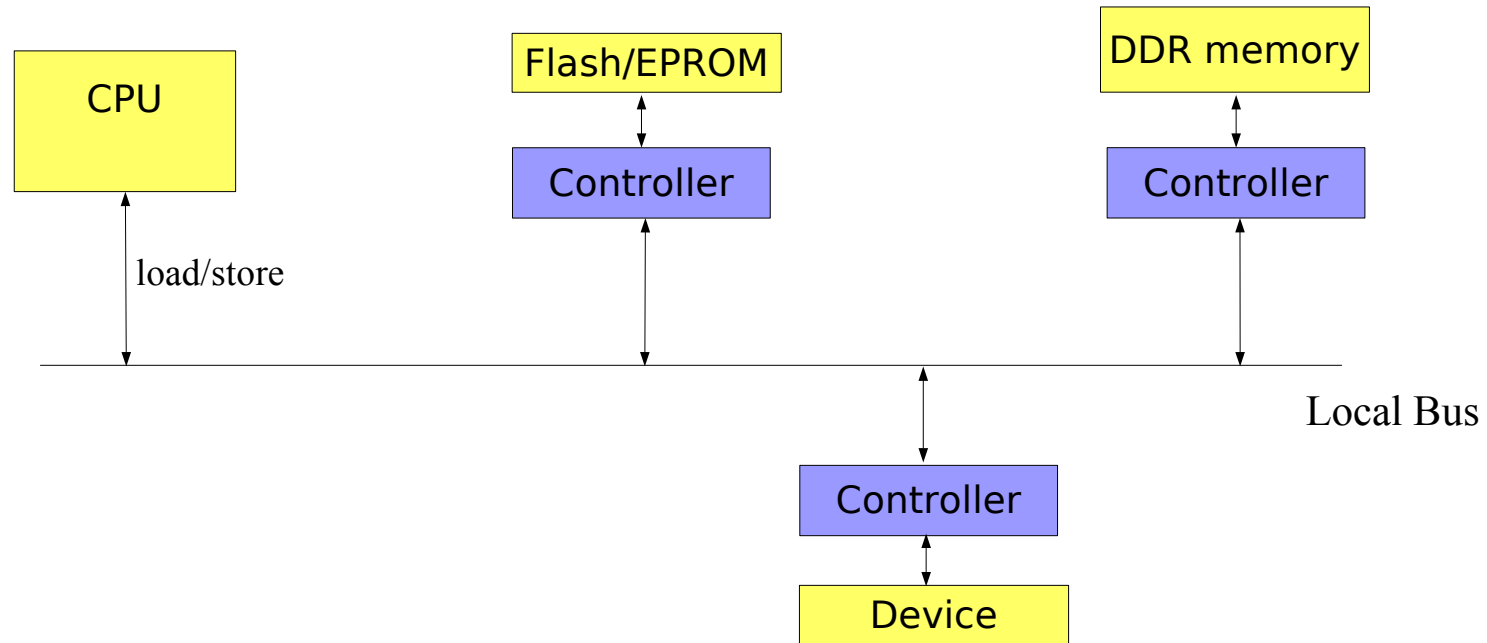
- Boot sequence – ARM example
 - Processor wakes up at a given address, at 0x0000-0000 (reset vector)
 - Starts executing there, but what is there?
- EPROM/Flash controller
 - Factory-configured bus to map EPROM at @0x0000-0000
- Memory Map
 - Each controller has a reserved address range

Hardware – Reset / Power-up



- Fetching the first instruction...
 - To execute an instruction, the processor has to fetch it → reset executes at 0x0000-0000
 - So start executing the code from the EPROM
 - Will start by configuring the DDR controller, give access to regular read-write memory (ram)
 - At some point, Flash/EPROM is re-mapped high, exposing ram @0x0000-0000
 - Software can then setup the IRQ/Trap vector @ 0x0000-0000

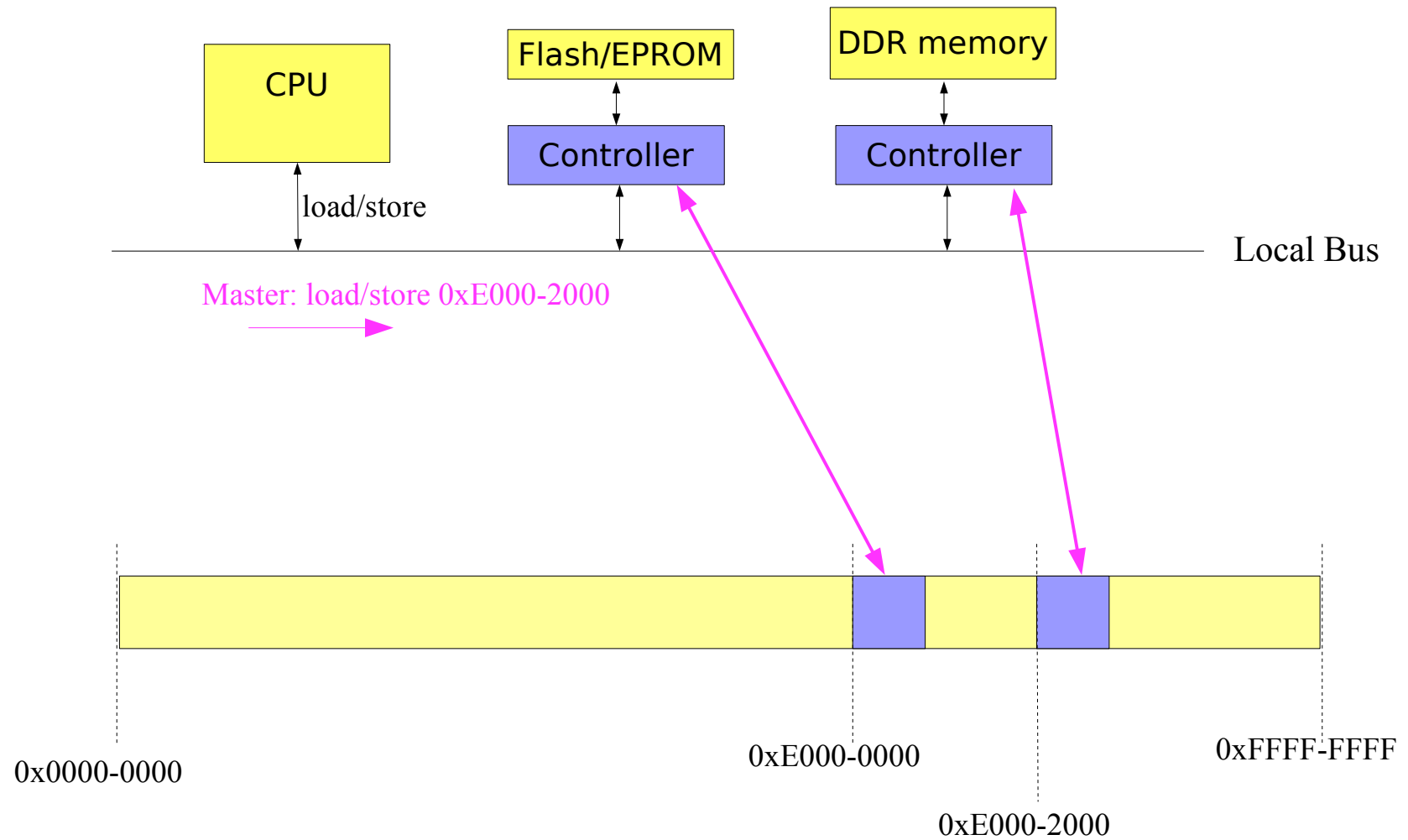
Hardware – Questions



How can software configure hardware controllers?

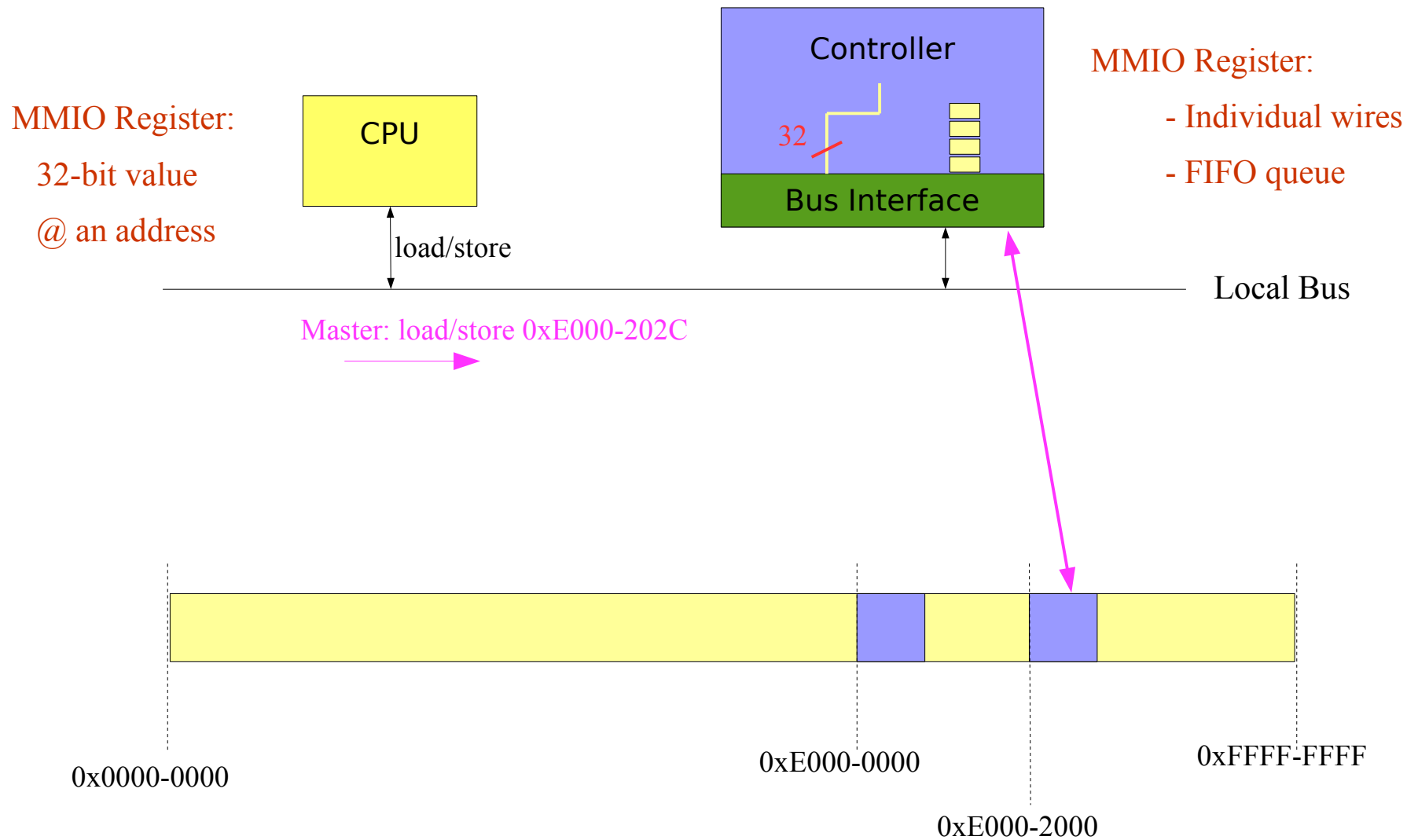
More generally, how does software interact with hardware controllers and thus with the devices attached to them?

Hardware – MMIO Registers



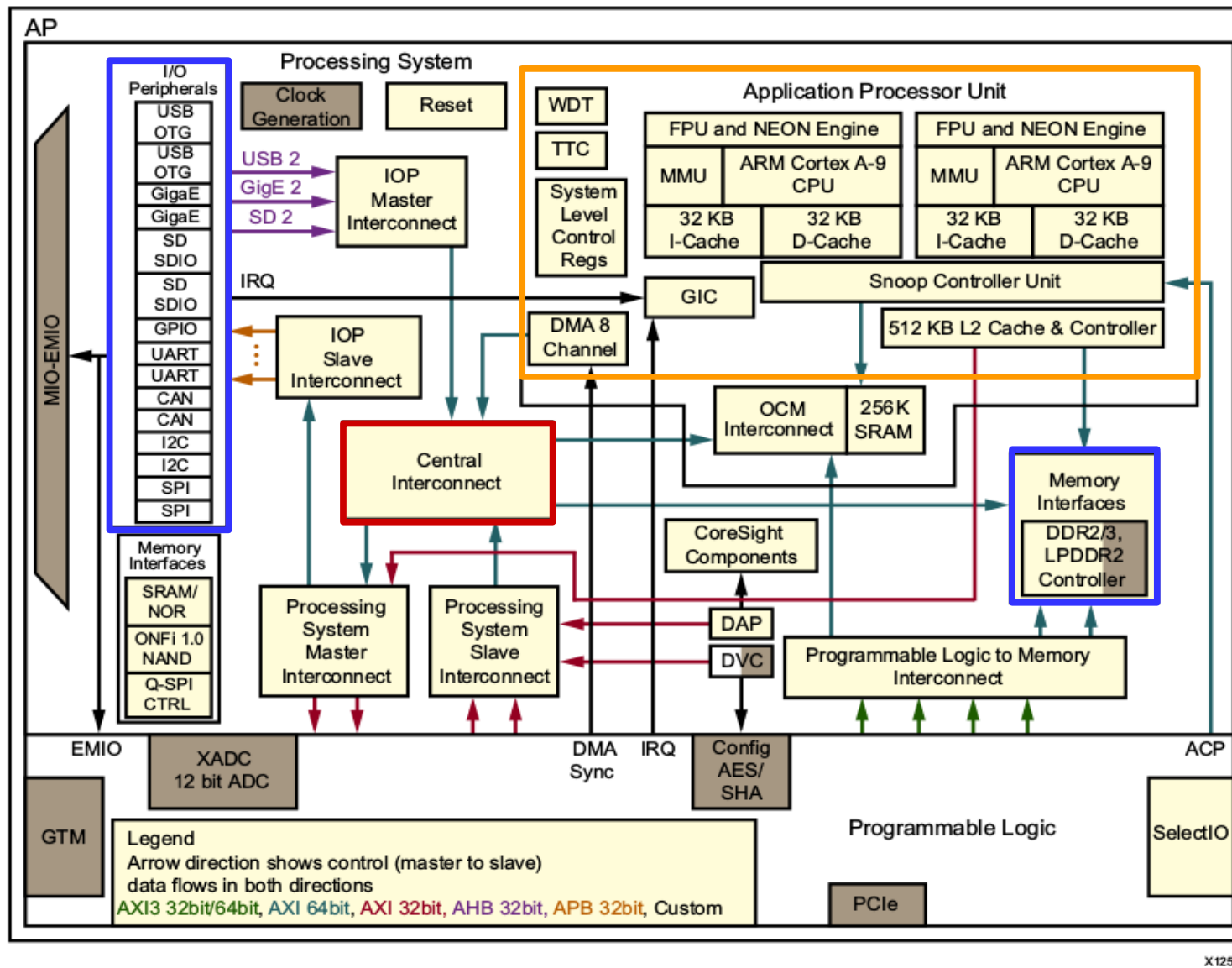
Memory-Mapped Input/Output Ranges

Hardware – MMIO Registers



MMIO: Memory-Mapped Input/Output

Hardware – Zynq-7000 Overview



Zynq-7000 Memory Map

Table 4-1: System-Level Address Map

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters ⁽¹⁾	Notes
0000_0000 to 0003_FFFF ⁽²⁾	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU ⁽³⁾
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF ⁽⁴⁾	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF ⁽²⁾	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Zynq-7000 Memory Map

Table 4-7: PS System Register Map

Register Base Address	Description (Acronym)	Register Set
F800_1000, F800_2000	Triple timer counter 0, 1 (TTC 0, TTC 1)	ttc.
F800_3000	DMAC when secure (DMAC S)	dmac.
F800_4000	DMAC when non-secure (DMAC NS)	dmac.
F800_5000	System watchdog timer (SWDT)	swdt.
F800_6000	DDR memory controller	ddrc.
F800_7000	Device configuration interface (DevC)	devcfg.
F800_8000	AXI_HP 0 high performance AXI interface w/ FIFO	afi.
F800_9000	AXI_HP 1 high performance AXI interface w/ FIFO	afi.
F800_A000	AXI_HP 2 high performance AXI interface w/ FIFO	afi.
F800_B000	AXI_HP 3 high performance AXI interface w/ FIFO	afi.
F800_C000	On-chip memory (OCM)	ocm.
F800_D000	eFuse ⁽¹⁾	-
F800_F000	Reserved	Reserved

Zynq-7000 Memory Map

Table 4-6: I/O Peripheral Register Map

Register Base Address	Description
E000_0000, E000_1000	UART Controllers 0, 1
E000_2000, E000_3000	USB Controllers 0, 1
E000_4000, E000_5000	I2C Controllers 0, 1
E000_6000, E000_7000	SPI Controllers 0, 1
E000_8000, E000_9000	CAN Controllers 0, 1
E000_A000	GPIO Controller
E000_B000, E000_C000	Ethernet Controllers 0, 1
E000_D000	Quad-SPI Controller
E000_E000	Static Memory Controller (SMC)
E010_0000, E010_1000	SDIO Controllers 0, 1

UART = Universal Asynchronous Receiver and Transmitter

Also called RS-232 (a standard for a serial line over 2 wires)

UART Device Example

UART... serial line controller, following the RS-232 protocol...

→ Essentially a FIFO and a status register...

Corresponding the **mmio** registers defined in the Zynq-7000/R1P8 Technical Reference Manual

```
#define UART_R1P8_CR          0x0000 /* UART Control Register */
#define UART_R1P8_MR          0x0004 /* UART Mode Register */
#define UART_R1P8_IER         0x0008 /* -- Interrupt Enable Register */
#define UART_R1P8_IDR         0x000C /* -- Interrupt Disable Register */
#define UART_R1P8_IMR         0x0010 /* -- Interrupt Mask Register */
#define UART_R1P8_ISR         0x0014 /* -- Channel Interrupt Status Register */
#define UART_R1P8_BAUDGEN     0x0018 /* Baude Rate Generator Register */
#define UART_R1P8_RXTOUT      0x001C /* -- Receiver Timeout Register */
#define UART_R1P8_RXWM        0x0020 /* -- Receiver FIFO Trigger Level Register */
#define UART_R1P8_MODEMCR     0x0024 /* -- Modem Control Register */
#define UART_R1P8_MODEMSR     0x0028 /* -- Modem Status Register */
#define UART_R1P8_SR          0x002C /* Channel Status Register */
#define UART_R1P8_FIFO        0x0030 /* Transmit & Receive FIFO */
#define UART_R1P8_BAUDDIV     0x0034 /* Baud Rate Divider Register */
#define UART_R1P8_FLOWD       0x0038 /* -- Flow Control Delay Register */
#define UART_R1P8_TXWM        0x0044 /* -- Transmitter FIFO Trigger Level Register */
```

UART Device Example

Interacting with a device:

- 1) choose one mmio range corresponding to your device
- 2) choose one or more register (at different offsets in that range)
- 3) work with one or more bits in that register

```
#define UART0 0xE0000000  
#define UART1 0xE0001000
```

```
#define UART_R1P8_SR          0x002C /* Channel Status Register */  
#define UART_R1P8_FIFO        0x0030 /* Transmit & Receive FIFO */  
/*
```

```
 * Channel Status Register (UART_R1P8_SR)  
 */
```

```
#define UART_R1P8_SR_TNFUL    (1 << 14)  
#define UART_R1P8_SR_TTRIG    (1 << 13)  
#define UART_R1P8_SR_FDELT    (1 << 12)  
#define UART_R1P8_SR_TACTIVE  (1 << 11)  
#define UART_R1P8_SR_RACTIVE  (1 << 10)
```

```
#define UART_R1P8_SR_TFUL     (1 << 4)  
#define UART_R1P8_SR_EMPTY    (1 << 3)  
#define UART_R1P8_SR_RFUL     (1 << 2)  
#define UART_R1P8_SR_REMPTY    (1 << 1)  
#define UART_R1P8_SR_RTRIG    (1 << 0)
```

UART – Initialization

```
void
uart_r1p8_init_regs(void* uart) { /* See Zynq TRM sequence (UG585 p598) */

    /* UART Character frame */
    mmio_reg_write32(uart,UART_R1P8_MR,UART_R1P8_MR_8n1);

    /* Baud Rate configuration */
    mmio_reg_setbits32(uart,UART_R1P8_CR,  UART_R1P8_CR_RXDIS | UART_R1P8_CR_TXDIS);
    mmio_reg_write32(uart,UART_R1P8_BAUDGEN, UART_R1P8_115200_GEN);
    mmio_reg_write32(uart,UART_R1P8_BAUDDIV, UART_R1P8_115200_DIV);
    mmio_reg_setbits32(uart,UART_R1P8_CR,  UART_R1P8_CR_RXRES | UART_R1P8_CR_TXRES);
    mmio_reg_setbits32(uart,UART_R1P8_CR,  UART_R1P8_CR_RXEN | UART_R1P8_CR_TXEN);

    /* Disable Rx Trigger level */
    mmio_reg_write32(uart,UART_R1P8_RXWM,  0x00);

    /* Enable Controller */
    mmio_reg_write32(uart,UART_R1P8_CR,  UART_R1P8_CR_RXRES | UART_R1P8_CR_TXRES |
        UART_R1P8_CR_RSTTO | UART_R1P8_CR_RXEN | UART_R1P8_CR_TXEN |
        UART_R1P8_CR_STPBRK);

    /* Configure Rx Timeout */
    mmio_reg_write32(uart,UART_R1P8_RXTOUT, 0x00);
    mmio_reg_write32(uart,UART_R1P8_IER, 0x00);
    mmio_reg_write32(uart,UART_R1P8_IDR,  UART_R1P8_IxR_ALL);

    /* No Flow delay */
    mmio_reg_write32(uart,UART_R1P8_FLOWD, 0x00);

    /* Deactivate flowcontrol */
    mmio_reg_clearbits32(uart,UART_R1P8_MODEMCR, UART_R1P8_MODEMCR_FCM);

    /* Mask all interrupts */
    mmio_reg_clearbits32(uart,UART_R1P8_IMR, 0x01FFF);
}
```

UART – Output

```
#define UART0 0xE0000000
#define UART1 0xE0001000
```

```
#define UART_R1P8_SR          0x002C /* Channel Status Register */
#define UART_R1P8_FIFO        0x0030 /* Transmit & Receive FIFO */
```

```
/*
 * Channel Status Register (UART_R1P8_SR)
 */
#define UART_R1P8_SR_TNFUL    (1 << 14)
#define UART_R1P8_SR_TTRIG    (1 << 13)
#define UART_R1P8_SR_FDELTA   (1 << 12)
#define UART_R1P8_SR_TACTIVE   (1 << 11)
#define UART_R1P8_SR_RACTIVE   (1 << 10)
#define UART_R1P8_SR_TFUL      (1 << 4)
#define UART_R1P8_SR_EMPTY     (1 << 3)
#define UART_R1P8_SR_RFUL      (1 << 2)
#define UART_R1P8_SR_EMPTY     (1 << 1)
#define UART_R1P8_SR_RTRIG     (1 << 0)
```

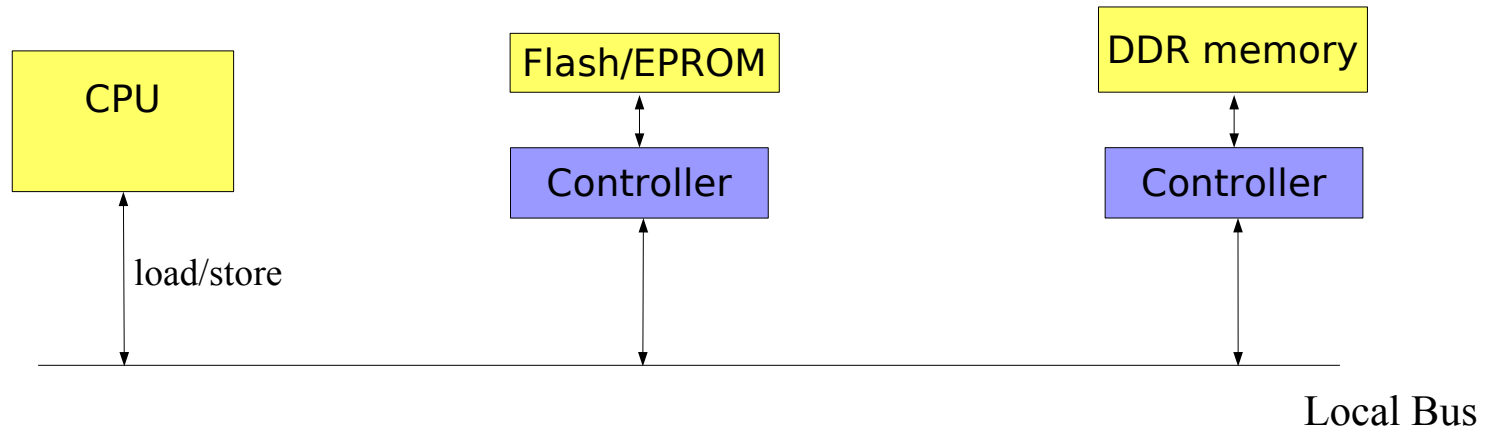
```
void
uart_r1p8_putc(void* uart, uint8_t c) {
    while((mmio_read32(uart, UART_R1P8_SR) & UART_R1P8_SR_TFUL) != 0)
        ;
    mmio_write32(uart, UART_R1P8_FIFO, c);
}
```


UART – Input

```
#define UART_R1P8_SR          0x002C /* Channel Status Register */
#define UART_R1P8_FIFO        0x0030 /* Transmit & Receive FIFO */
/*
 * Channel Status Register (UART_R1P8_SR)
 */
#define UART_R1P8_SR_TNFUL    (1 << 14)
#define UART_R1P8_SR_TTRIG    (1 << 13)
#define UART_R1P8_SR_FDELT    (1 << 12)
#define UART_R1P8_SR_TACTIVE  (1 << 11)
#define UART_R1P8_SR_RACTIVE  (1 << 10)
#define UART_R1P8_SR_TFUL     (1 << 4)
#define UART_R1P8_SR_TEMPTY    (1 << 3)
#define UART_R1P8_SR_RFUL     (1 << 2)
#define UART_R1P8_SR_EMPTY    (1 << 1)
#define UART_R1P8_SR_RTRIG    (1 << 0)
```

```
uint8_t
uart_r1p8_getc(void* uart){
    while((mmio_read32(uart, UART_R1P8_SR) & UART_R1P8_SR_EMPTY))
        ;
    return mmio_read32(uart, UART_R1P8_FIFO);
}
```

Hardware – Questions



How can software configure hardware controllers?

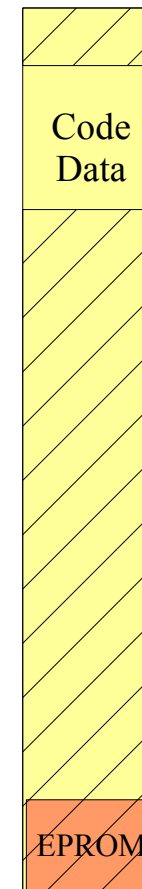
How does the boot process go on?

Boot Process

- **Hardware wakes up after reset**
 - Execute from EPROM/Flash, often called firmware
 - In the Intel world, it is called the BIOS
- **Bare-metal software**
 - Sometimes, there is nothing else than the firmware
 - Like for a small IoT sensor or your coffee machine
 - Otherwise, the boot process goes on...

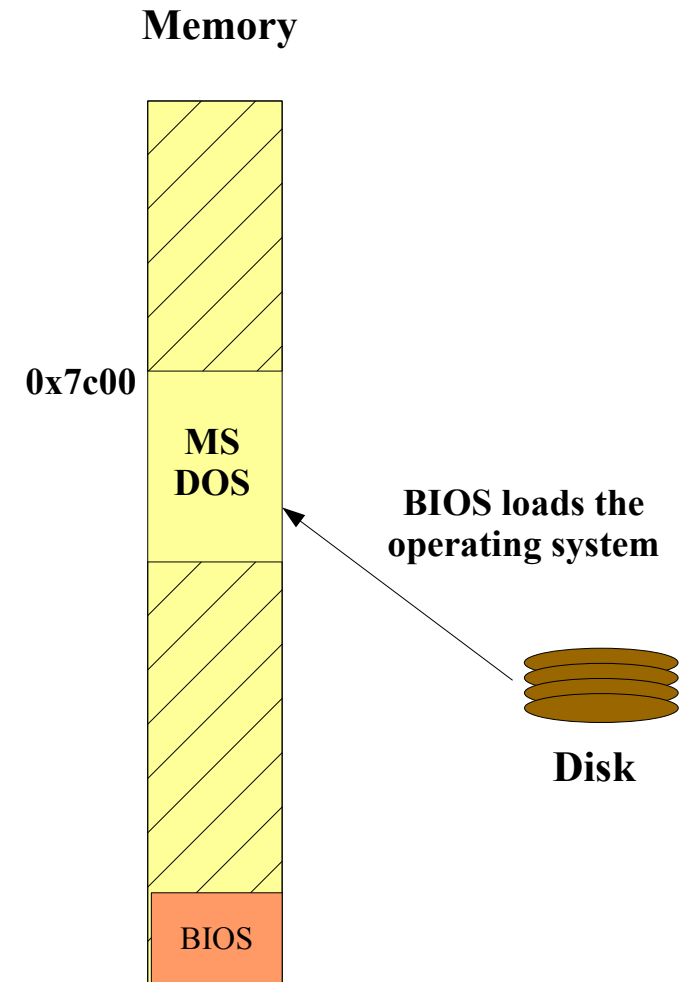
Let's discuss the Intel world...

Memory



Boot Process

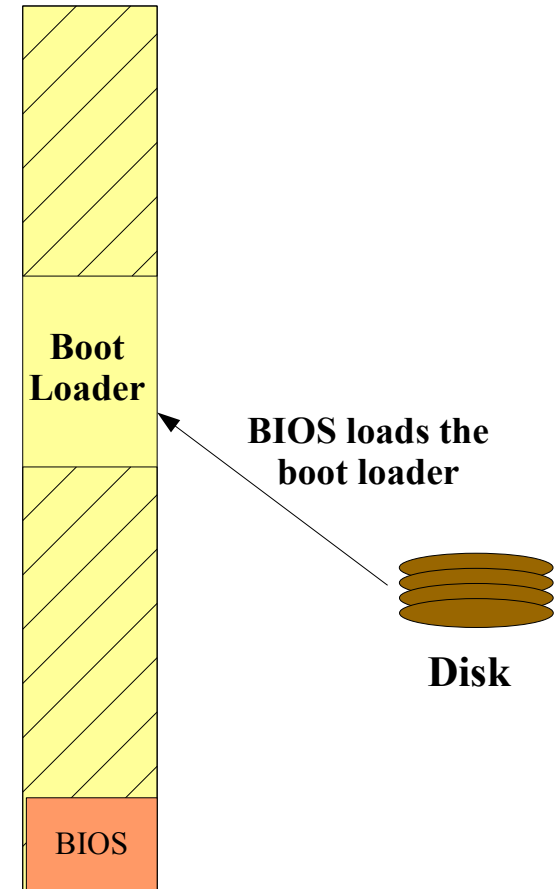
- BIOS is resident in memory
 - Initializes enough devices (SD card, hard disks, etc.)
 - Provides Basic Input/Output Services (BIOS)
- The old days of MS-DOS
 - Microsoft DOS was loaded by the BIOS
 - It was the operating system
 - Essential a file system layer on top of the BIOS
 - The format was the FAT



Boot Process

- BIOS is resident in memory
 - Too many devices... not enough flexibility...
 - Here comes the boot loader
- Boot loader is the next stage in the boot process
 - Usually means more functionalities than the BIOS
 - It is also the first stage that is more portable
 - Relies on the hardware initialization done by the BIOS
- Boot loader offers the ability to
 - Choose what to load next...
 - And from where it is loaded from...

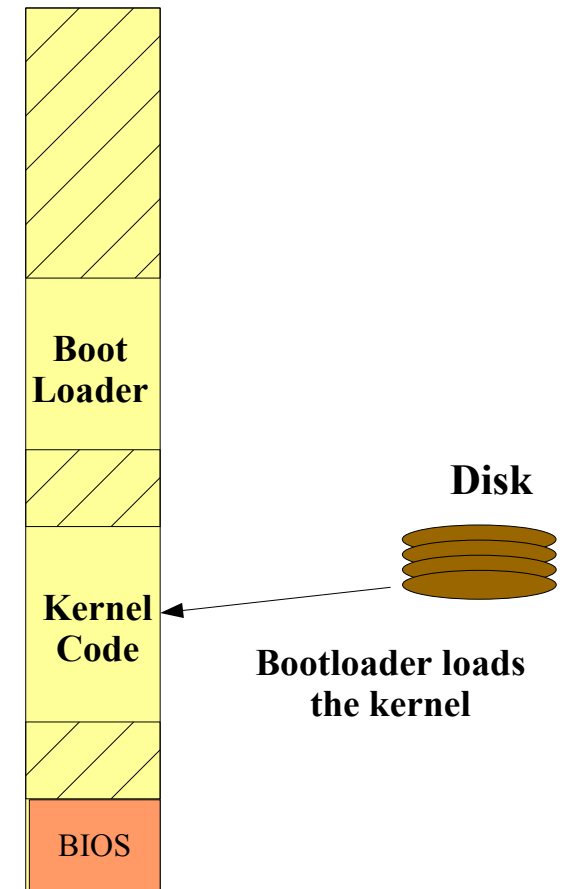
Memory



Boot Process

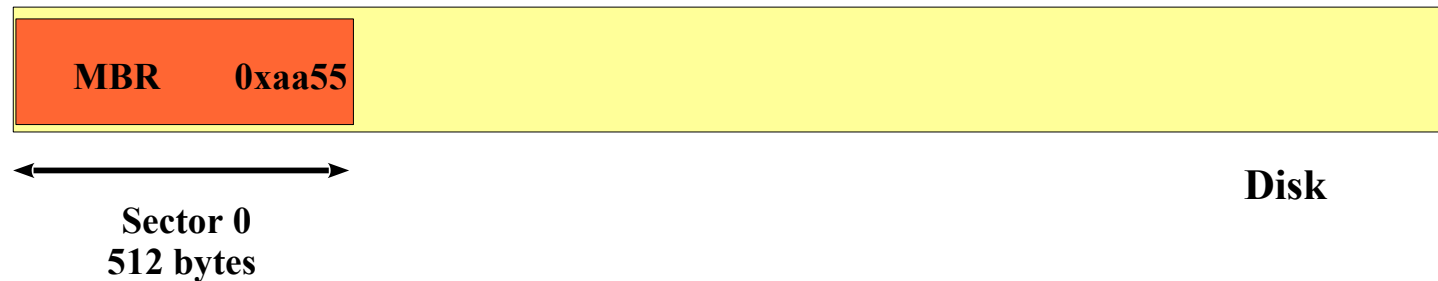
- The kernel is the next state in the boot process
 - The kernel takes over the machine
 - The boot loader is no longer needed
 - But the BIOS remains, it may be used or not
- Which kernel?
 - Could be the Linux kernel...
 - Or it could be you own bare-metal software

Memory



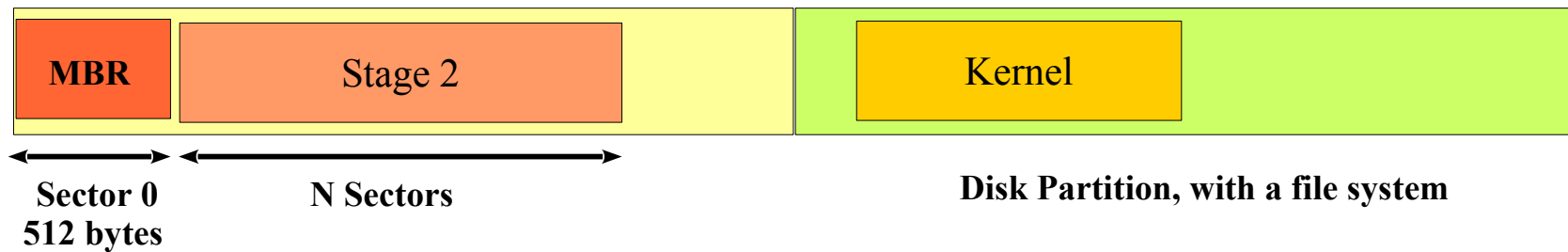
X86 Boot Sequence

- BIOS finds a bootable device
 - Master Boot Record (first disk sector)
 - Only bootable if MBR ends with 0xaa55 (at offset 0x1fe)
 - Loaded by the BIOS at 0x7c00
 - Starts executing at 0x7c00, but in 16bit legacy mode



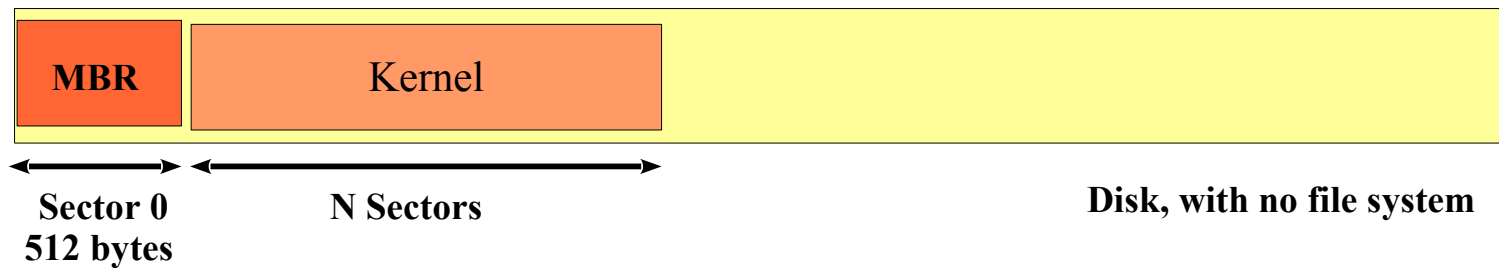
X86 Boot Sequence

- **Bootloader**
 - Usually has multiple stages, because the MBR is really small
 - Stage1 is the MBR
 - Stage2 is the rest of the boot loader
 - Stage1
 - Describes the partitions and contains a small amount of code: loading the stage2
 - Stage2
 - Loads the kernel from a partition with a file system
 - Must therefore have the code to understand that file system structure on disk



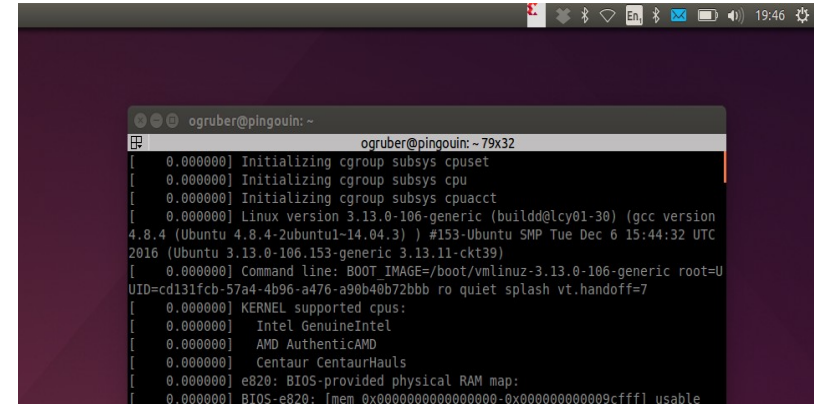
X86 Boot Sequence

- Our own boot loader
 - The MBR is enough, loads our kernel
- Our own kernel
 - Small and simple bare-metal code

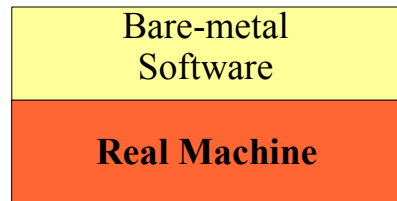


Development Environment

- Using a real board
 - Relies on using a USB-Serial cable
- Through a JTAG-USB port
 - JTAG to upload the firmware
 - JTAG hardware and software debugging
 - Serial line for a command-line interface



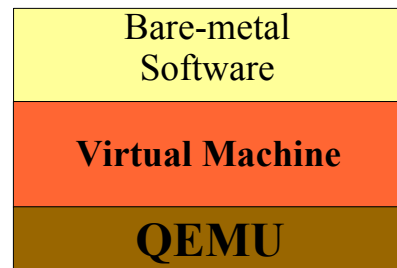
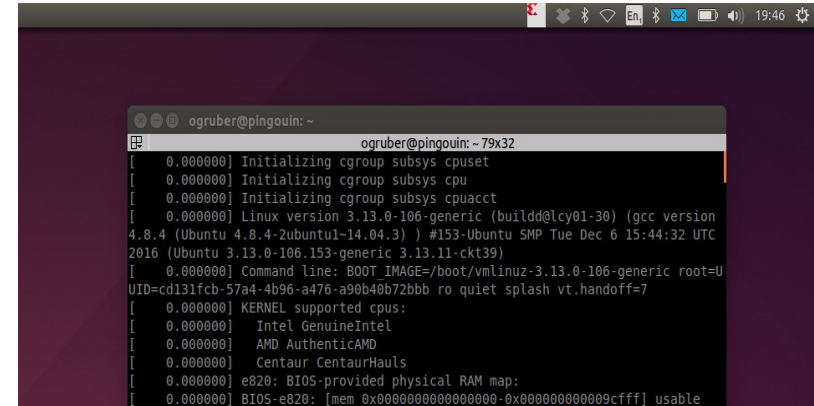
```
ogruber@pinguin: ~ 79x32
[ 0.000000] Initializing cgroup subsys cpuset
[ 0.000000] Initializing cgroup subsys cpu
[ 0.000000] Initializing cgroup subsys cpuacct
[ 0.000000] Linux version 3.13.0-106-generic (build@lcy01-30) (gcc version
4.8.4 (Ubuntu 4.8.4-2ubuntu1-14.04.3) ) #153-Ubuntu SMP Tue Dec 6 15:44:32 UTC
2016 (Ubuntu 3.13.0-106.153-generic 3.13.11-ckt39)
[ 0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-3.13.0-106-generic root=U
UID=c0131fcb-57a4-4b96-a476-a90b40b72bbb ro quiet splash vt.handoff=7
[ 0.000000] KERNEL supported cpus:
[ 0.000000] Intel GenuineIntel
[ 0.000000] AMD AuthenticAMD
[ 0.000000] Centaur CentaurHauls
[ 0.000000] e820: BIOS-provided physical RAM map:
[ 0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009cfff] usable
```



USB – Serial Cable – RS 232

Development Environment

- Using an emulator – QEMU
 - A regular Linux process
 - Emulates a machine for your bare-metal software
 - Direct support for GDB debugging
 - Serial line for a command-line interface

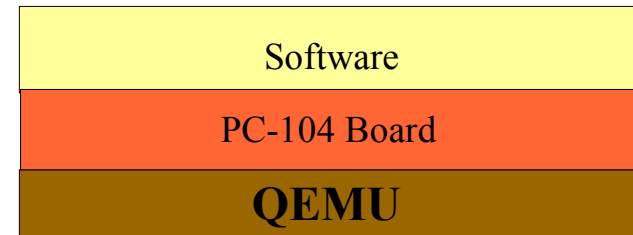


USB – Serial Cable – RS 232

QEMU

- In-Process Emulator
 - A virtual machine...
 - Looks real to your software...
- QEMU is a regular Linux process

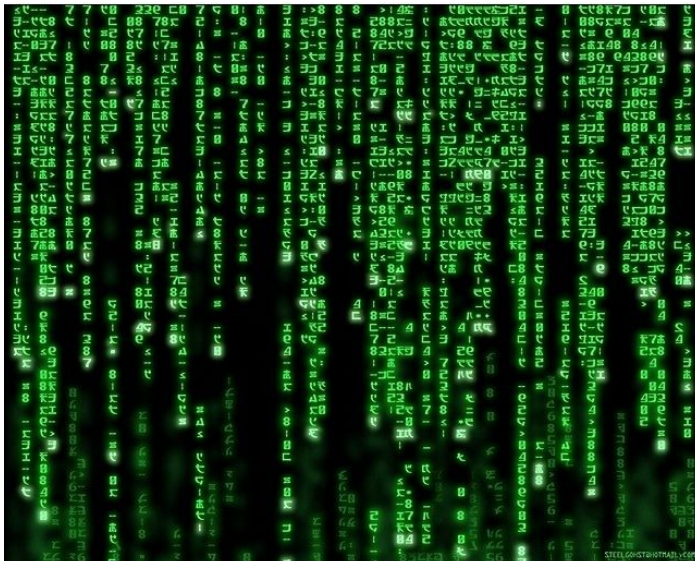
```
$ qemu-system-i386 -hda disk.img -serial mon:stdio
```



Your Target Platform



PC-104 Board
→ i386
→ serial line ~ COM1

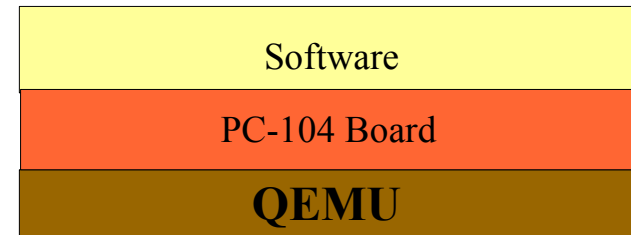


"The matrix is a prison you cannot see, taste, or smell." Morpheus

- In-Process Emulator

- PC-104 Board emulation
- Like an old PC from the 80's

\$ qemu-system-i386 -hda disk.img -serial mon:stdio



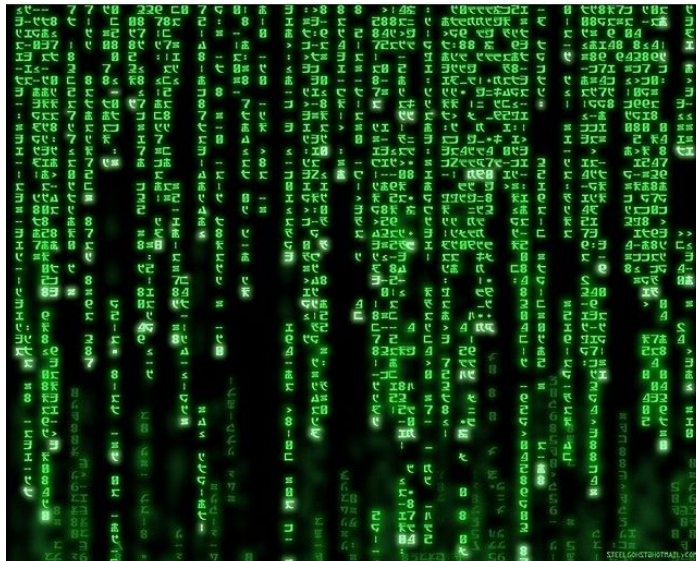
Your Target Platform



UART: COM1
Base Address: 0x3F8

Status: 0x3F8 + 0x05
Bit 0x20 → can write a character
Bit 0x01 → can read a character

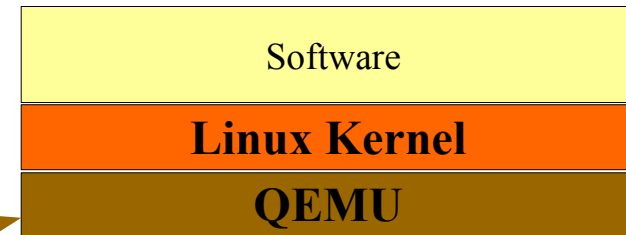
Out register: 0x3F8
In register: 0x3F8
IRQ: 4



QEMU

- In-Process Emulator

- QEMU monitor



```
$ qemu-system-i386 -serial mon:stdio
```

```
Crtl-A c  
(qemu) info qtree
```

```
...  
dev: i440FX-pcihost, id ""  
  irq 0  
  bus: pci.0  
  dev: PIIX3, id ""  
    class ISA bridge, addr 00:01.0,  
      pci id 8086:7000 (sub 1af4:1100)  
    bus: isa.0  
    type ISA  
    dev: isa-serial, id ""  
      index = 0 (0)  
      iobase = 1016 (0x3f8)  
      irq = 4 (0x4)  
      chardev = "serial0"  
      wakeup = 0 (0)  
      isa irq 4
```

Your Target ↔ **Platform**



UART: COM1
Base Address: 0x3F8

Status: 0x3F8 + 0x05
Bit 0x20 → can write a character
Bit 0x01 → can read a character

Out register: 0x3F8
In register: 0x3F8
IRQ: 4

What's next?

- Hands-on Learning
 - The worklog in Standalone...