M2PGi – M2M

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

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

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

This Course – Two Parts

- Part One – Pr. Olivier Gruber – 50%
  - 5 weeks, widening your horizon as developers
  - Wandering outside of the Linux world
  - Also a primer before part two
- Part Two – Pr. Didier Donsez – 50%
  - 5 weeks, experimenting with an IoT infrastructure
  - Sensors → Gateway → Data-gathering Server

Machine to Machine

Freely-adapted excerpts from Wikipedia:

Machine to machine (M2M) is direct communication between devices using any communications channel, including wired and wireless.

M2M communication can include industrial instrumentation, enabling sensors to communicate the information it records to software systems that can use it.

The Internet of things (IoT) describes the network of physical objects—"things"—that embed sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and software systems over the Internet.

Traditional fields of embedded systems, wired or wireless sensor networks, control systems, automation (including home and building automation), and many others all contribute to enabling the Internet of things.

IoT reference architecture

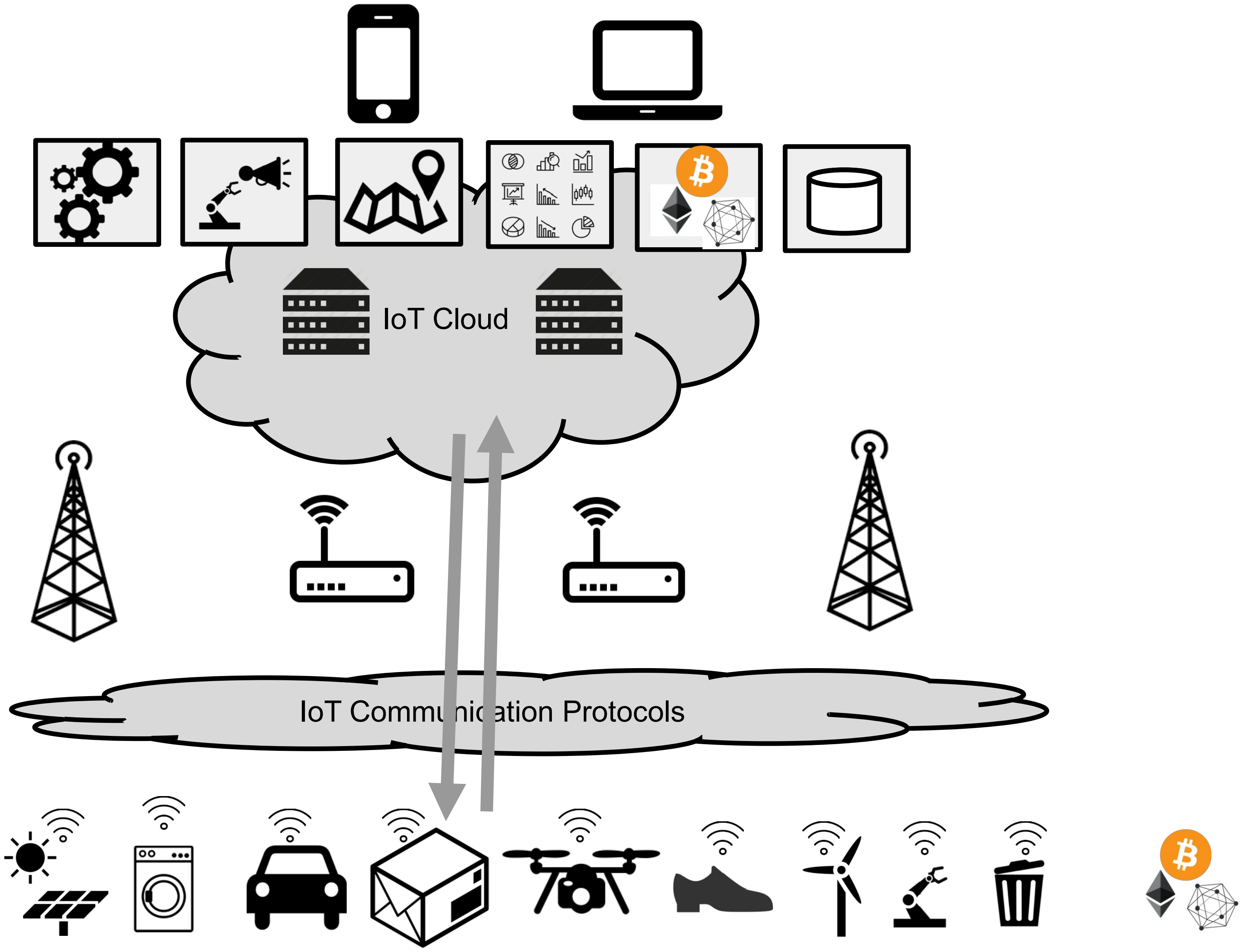
IoT Applications

Cloud infrastructure
(public, private)

Fog/Edge Computing

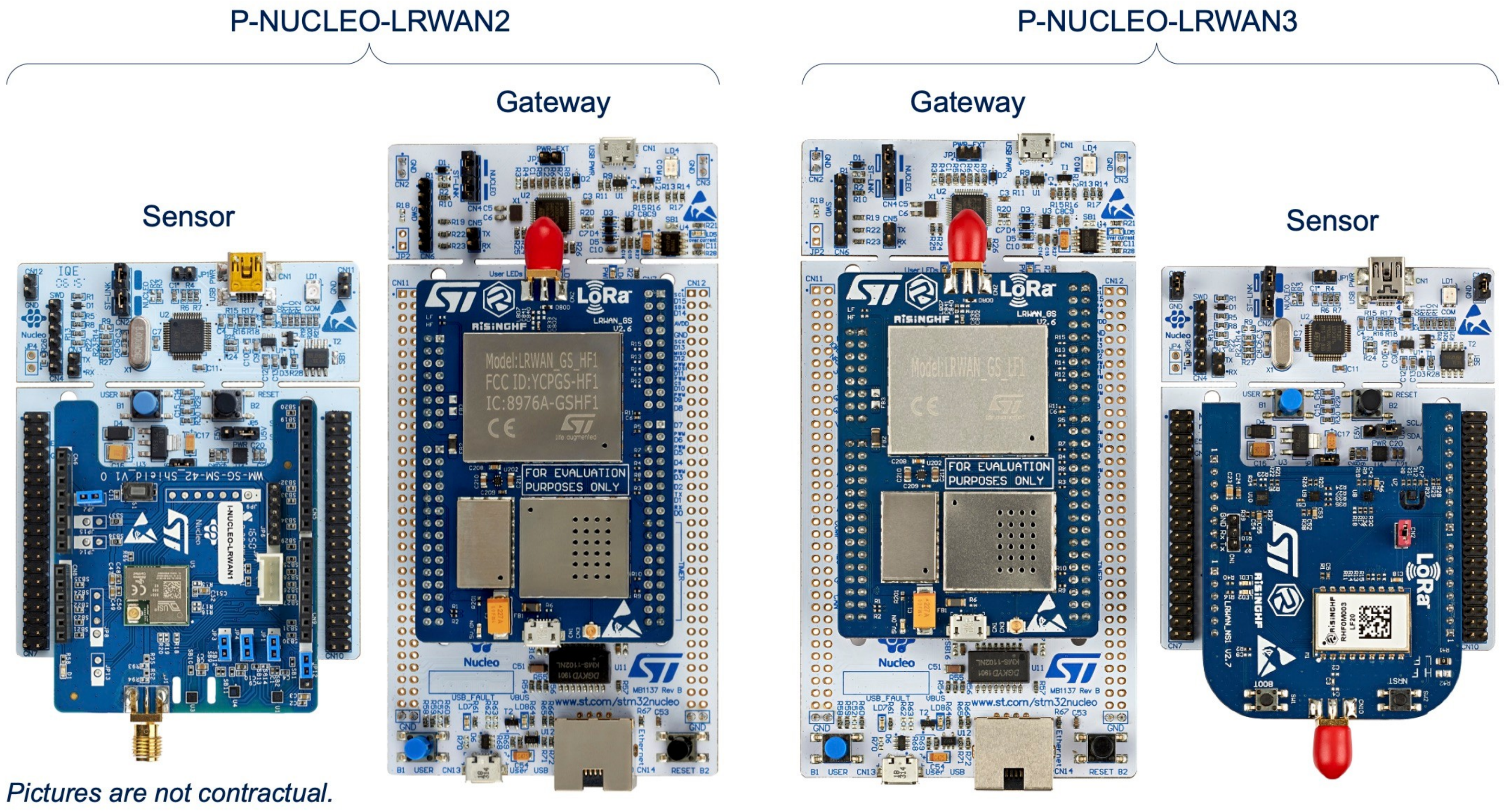
Communications
- wired/wireless
- IP / No IP
- licensed/free bands

Connected Things
(sensors & actuators)



P-Nucleo-LRWAN Starter Packs

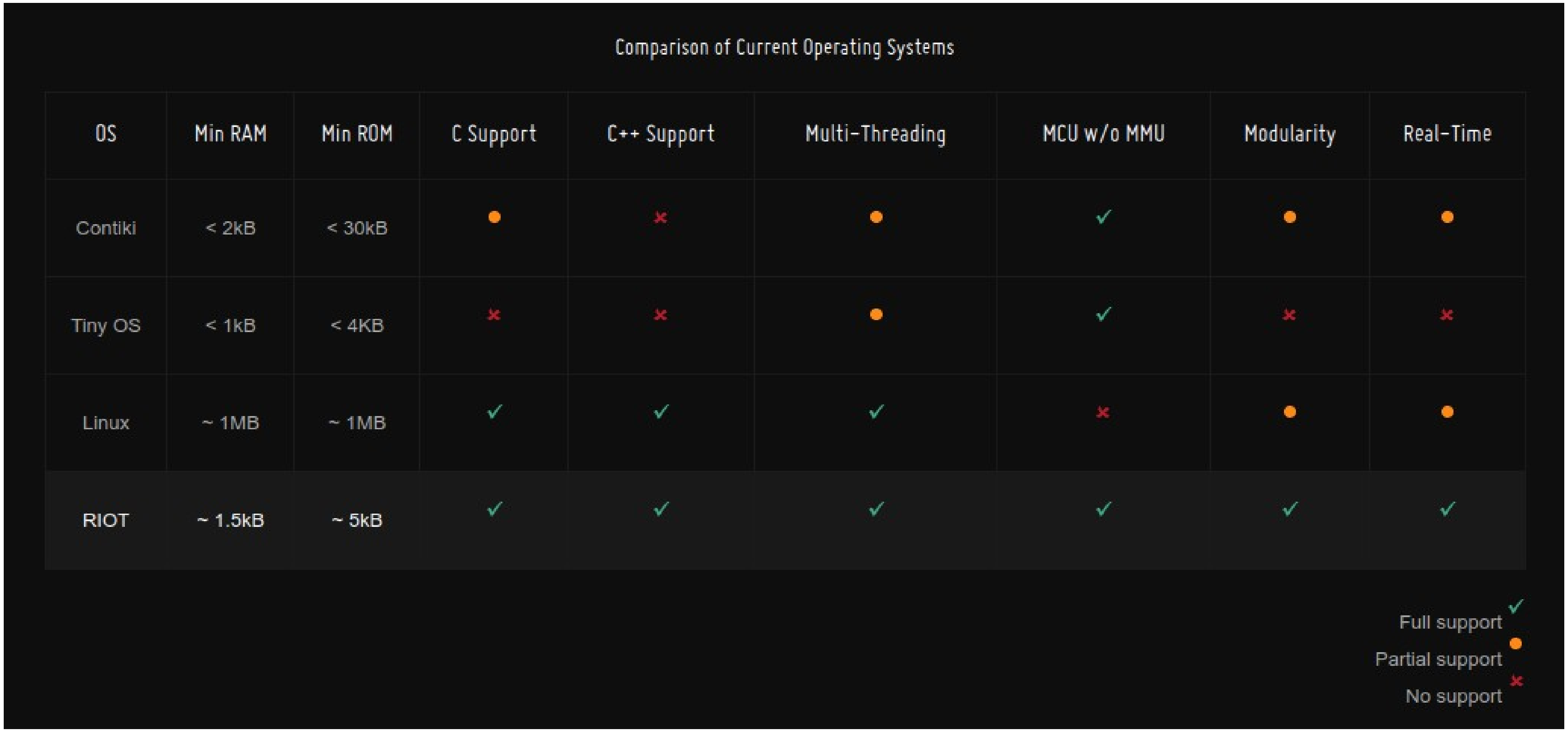
ToDo: Pick up your P-Nucleo-LRWAN2 at the fablab, after January 30th, but before the lectures start with Pr. Didier Donsez.



Software – RIOT

The friendly operating system for the internet of things.

RIOT is free open-source, developed by a grassroots community



Comparison of Current Operating Systems

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	< 2kB	< 30kB	●	✗	●	✓	●	●
Tiny OS	< 1kB	< 4KB	✗	✗	●	✓	✗	✗
Linux	~ 1MB	~ 1MB	✓	✓	✓	✗	●	●
RIOT	~ 1.5kB	~ 5kB	✓	✓	✓	✓	✓	✓

Full support	✓
Partial support	●
No support	✗

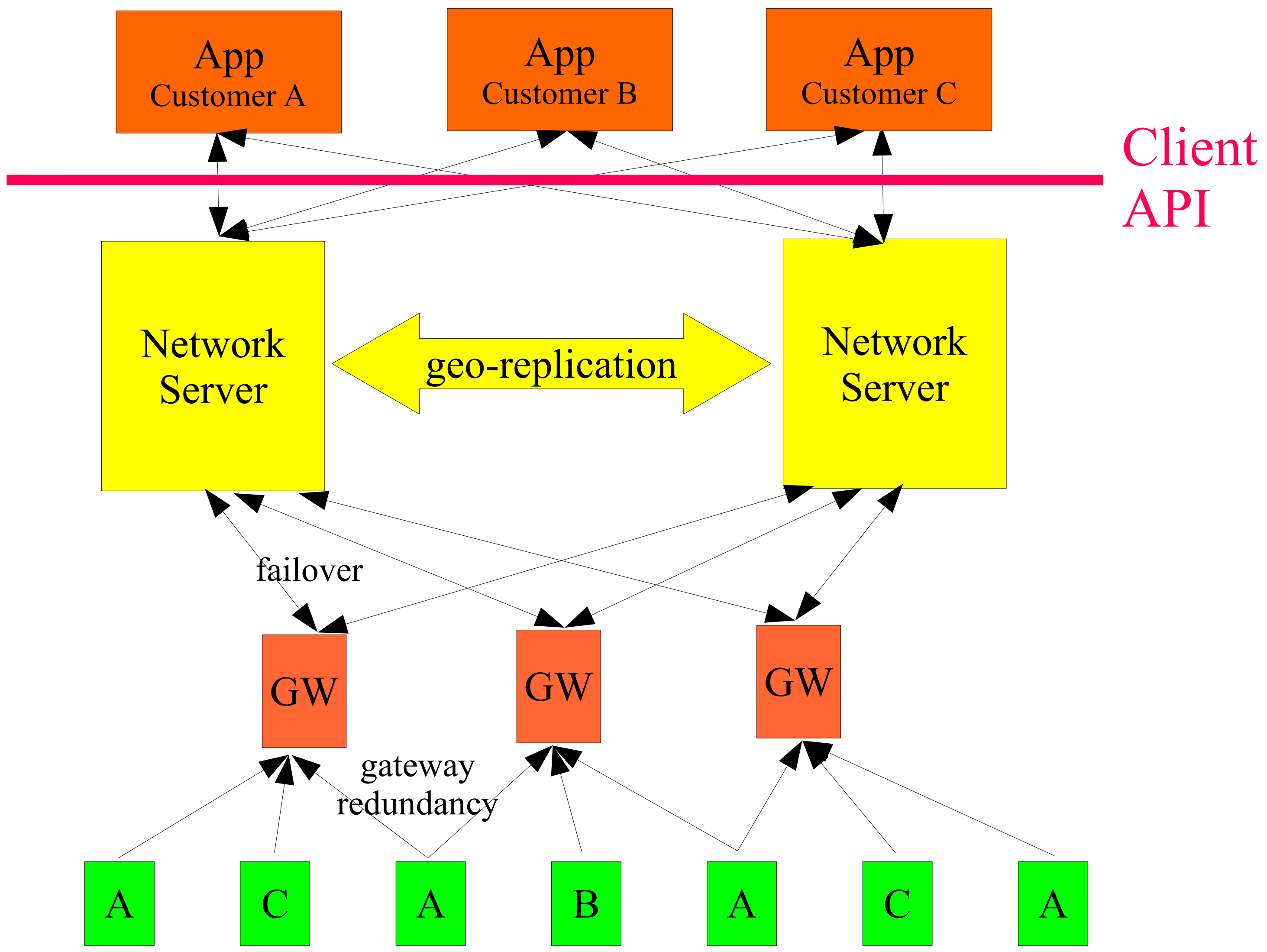
IoT Reference Architecture (ii)

Applications
Long Term Storage
(SaaS/On premise)

Core Network
* highly available
* high performance
* transient storage
(Cloud/On premise)

Base Stations
/ Gateways
(Edge/Fog)

Fixed Endpoints
Mobile Endpoints



First Part – Overview

- Widening your horizon as a developer
  - What you probably know about...

Probably application programming, right?

C, Java, JavaScript, Python...

Basic usage of your operating system, right?

File system and Graphical User Interface
Maybe a hint of shell usage/scripting...

How do we program these?

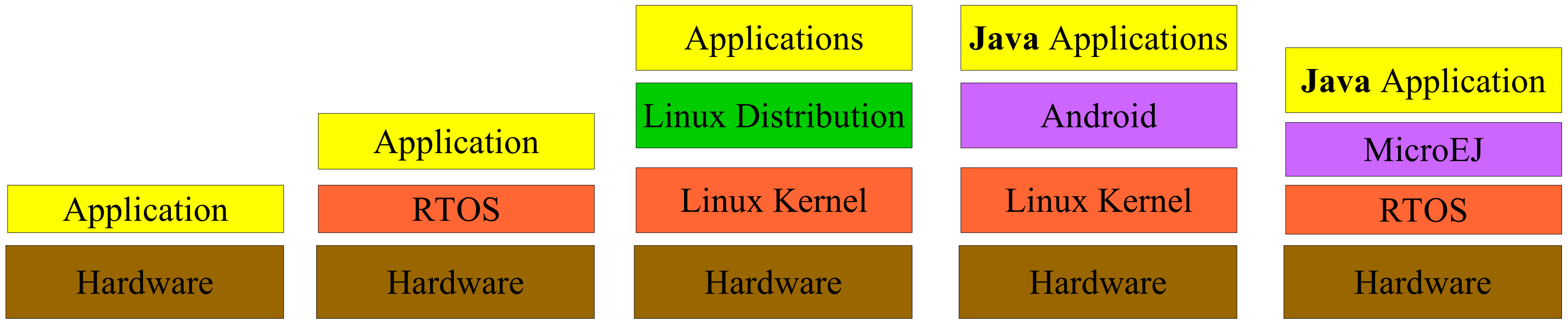
Which tools do we use?

First Part – Overview

Bare-metal programming
- Low-level C programming
- Lucky if using an Hardware Abstraction Layer (HAL)
- Otherwise needs some assembly language programming

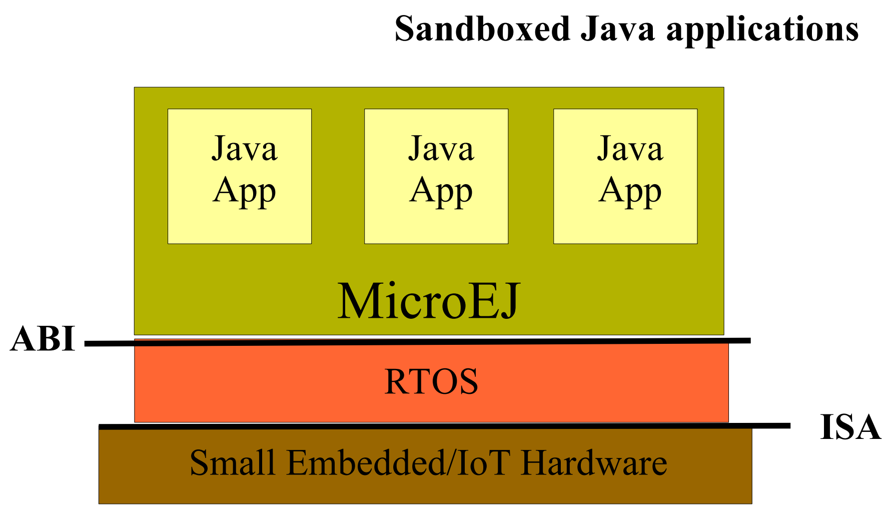
RTOS programming
- Higher-level programming… but not the usual Linux
- Timing-aware programming...

Java programming
- Increasingly more Java in the embedded world...



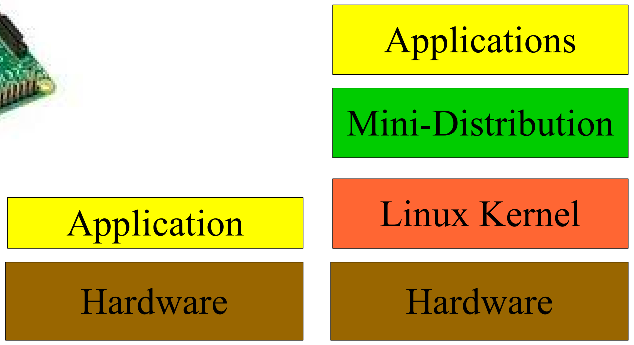
French Tech – MicroEJ

- Java for small and smart IoT devices
  - Similar virtualization technology as Android but *tighter* implementation
    - Android: $15 processor, 32MB of memory
    - MicroEJ: $1 processor, 128KB of memory
  - Google Cloud IoT Partners
    - IoT solutions that leverage Google's secure, global, and scalable infrastructure

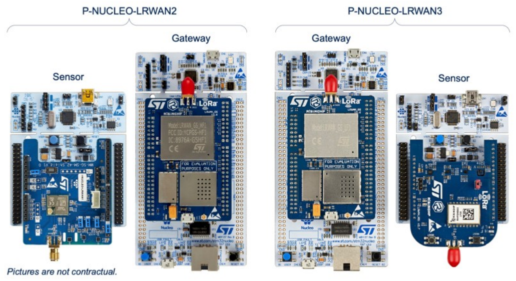


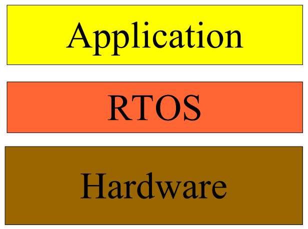
This Course Summary

Part-I Bare-metal programming
- Bare-metal programming
- Making your own minimal Linux distribution



Part-II RTOS/IoT programming
- RTOS programming
- IoT Infrastructure





First Part – Pedagogy

- Inverted pedagogy --- Team learning, individual work
  - Guided hands-one learning and coding
  - A work log to read, to understand, to complete, and to make your own
- The work log is for yourself first
  - A document about what you did and what you have learned
  - This document is first and foremost for you own records
- The destination is not the goal, the goal is the learning along that path
  - Don‘t just answer the questions…
  - Don‘t just read and understand...
  - Learn the demonstrated know-how… put in practice...
  - Discuss what you have learned/understood
  - Explain to others – Ask questions

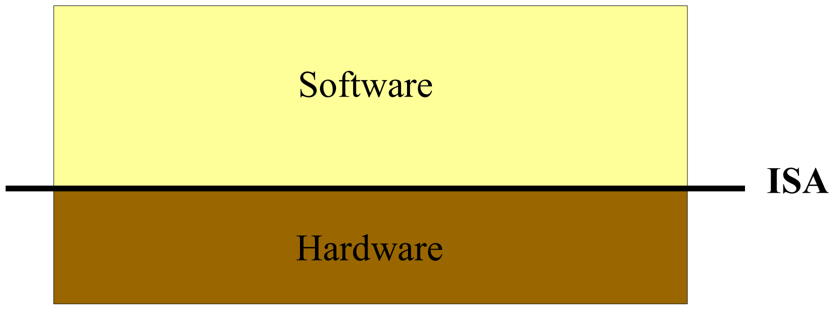
First Part – Evaluation

- Part-I Evaluation
  - No exam – but <u>weekly progress checkpoints</u>
  - <u>Every week:</u> you will surrender your work (work log and your code)
  - Your <u>weekly involvment</u> is the largest part of your final grade

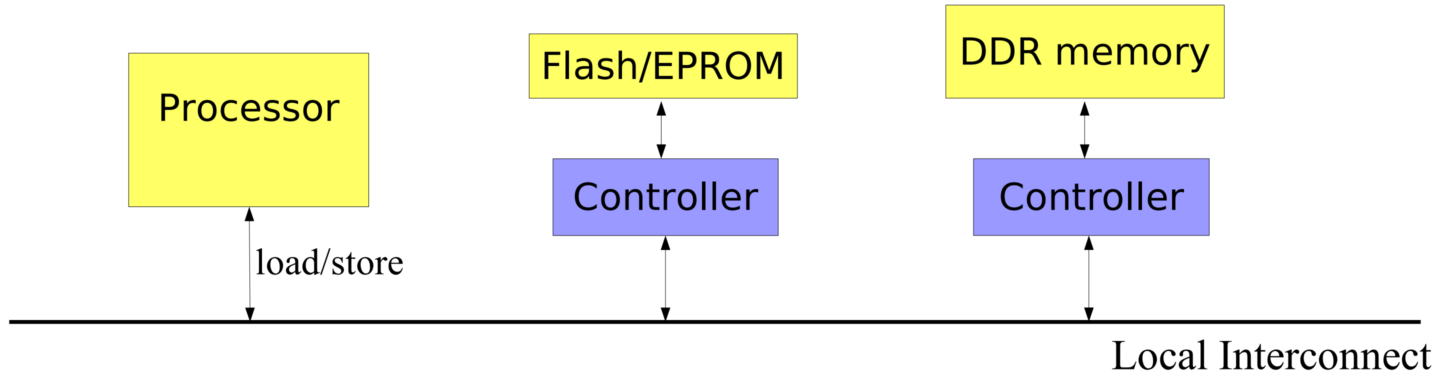
Bare-metal Development

- 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.
- Cross-development Toolchain
  - Compiler, linker, debugger, and other tools

```
$ sudo apt-get install gcc-arm-none-eabi   gdb-arm-none-eabi   binutils-arm-none-eabi
```

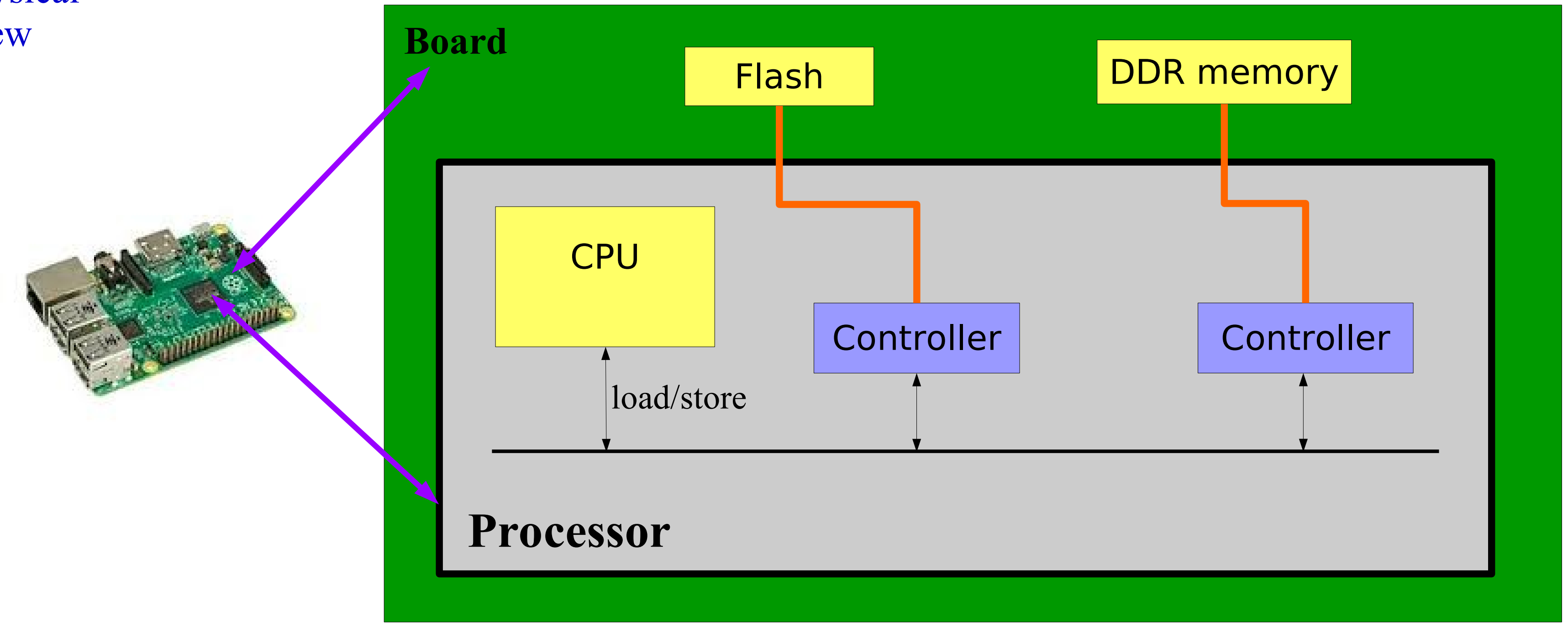


Hardware – Basics

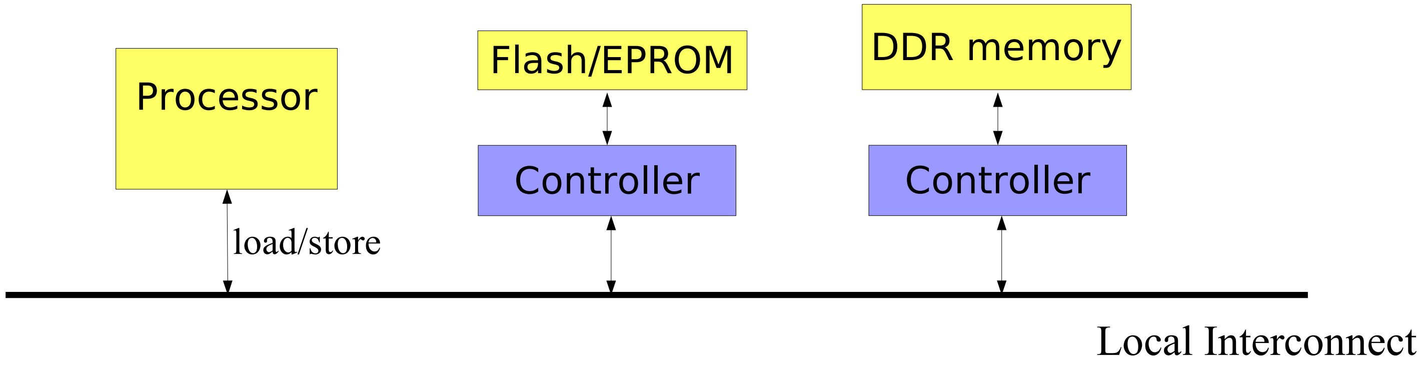


Conceptual View

Physical View

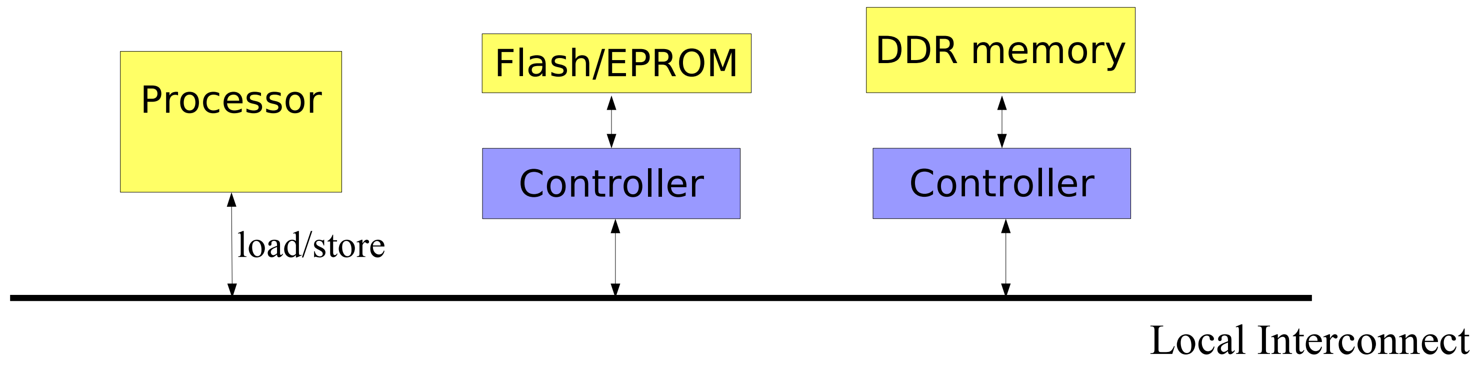


Hardware – Reset / Power-up



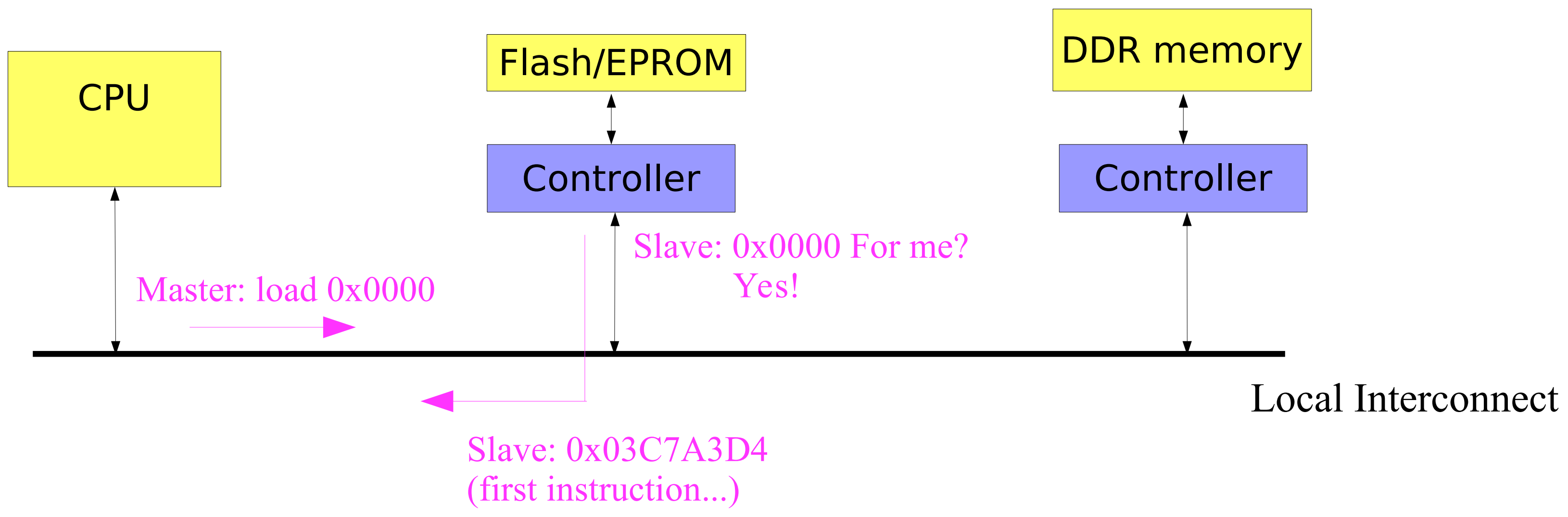
- Processor
  - Only knows how to issue load/store operations on the bus
  - Forever loop: *fetch – decode – issue*
- Local Interconnect (also called bus)
  - Data wires + control wires
  - Routes load/store operations to controllers

Hardware – Reset / Power-up



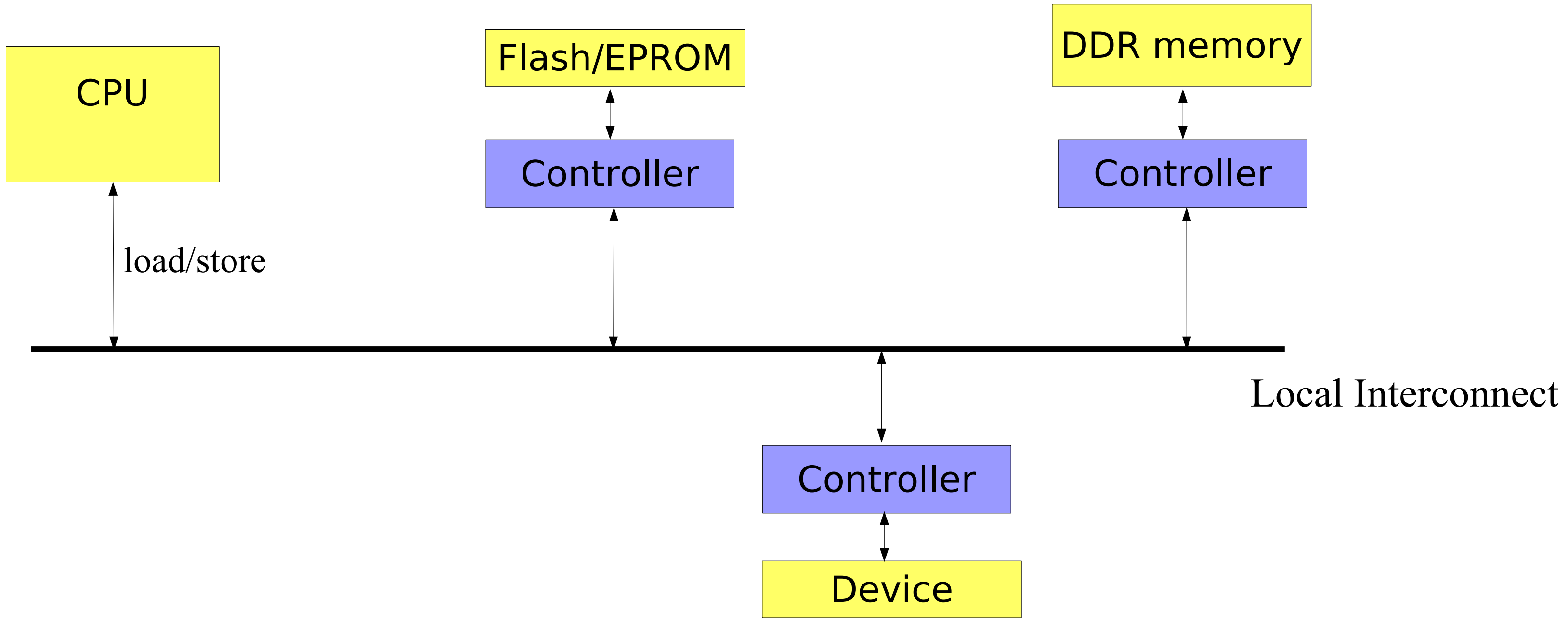
- Boot sequence – ARM example
  - Processor wakes up at a given address, at 0x0000-0000 (reset vector)
  - Starts executing there, but what is there?
- EPROM/Flash controller
  - Factory-configured bus to map EPROM at @0x0000-0000
- Memory Map
  - Each controller has a reserved address range
  - Some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

Hardware – Reset / Power-up



- Fetching the first instruction...
  - Reset the processor sets the program counter register to 0x0000-0000
  - So start executing the code from the EPROM
  - May stay executing in EPROM or may setup the execution in RAM
  - May remap the Flash/EPROM, exposing RAM @0x0000-0000
  - Sofware can then setup the IRQ/Trap vector @ 0x0000-0000

Hardware – Questions

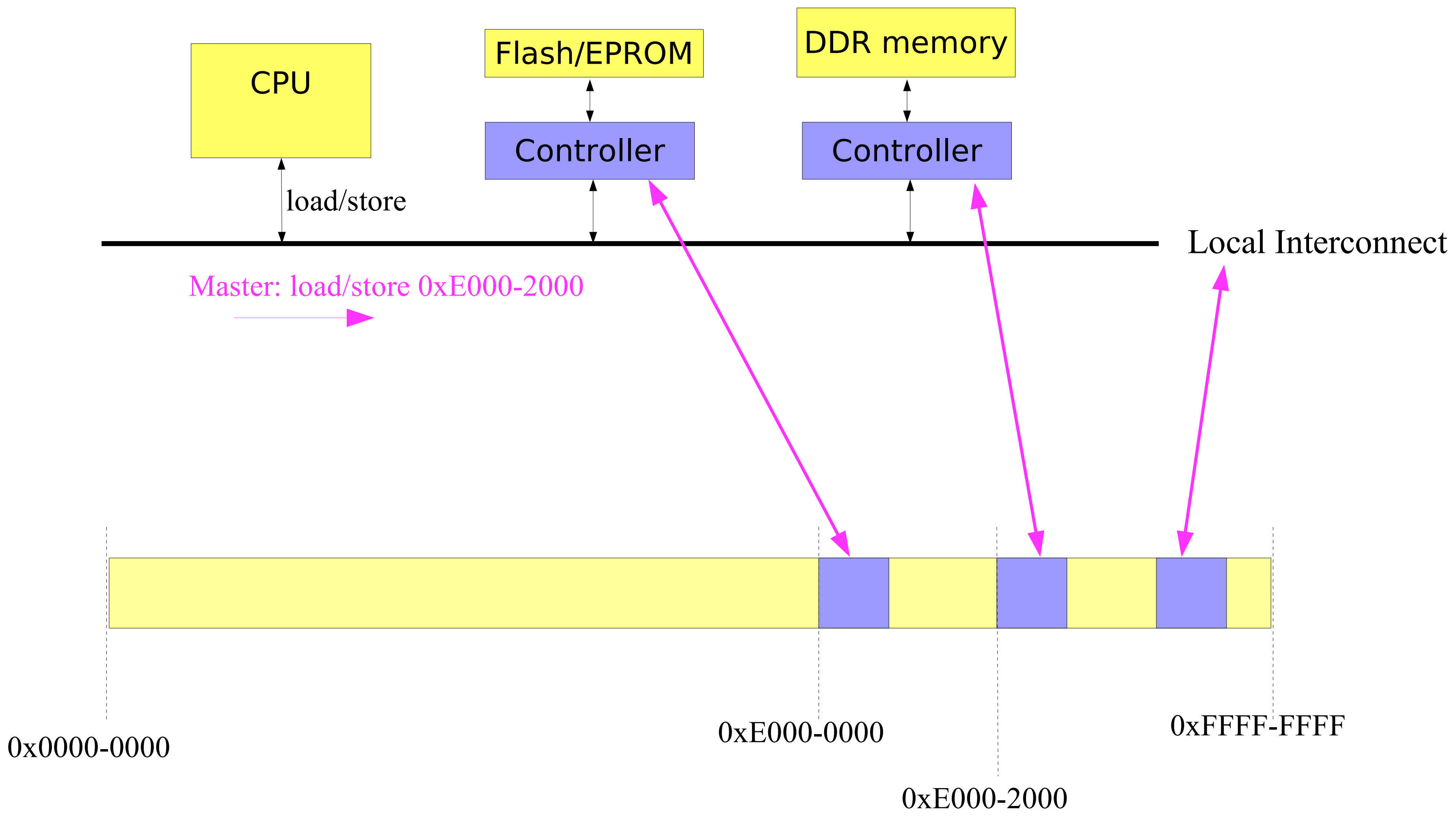


How can the Flash/EPROM can be remaped?

How can software configure hardware controllers?

More generally, how does software interacts with devices attached to hardware controllers?

Hardware – MMIO Registers



Memory-Mapped Input/Output Ranges

Hardware – MMIO Registers

MMIO Register:

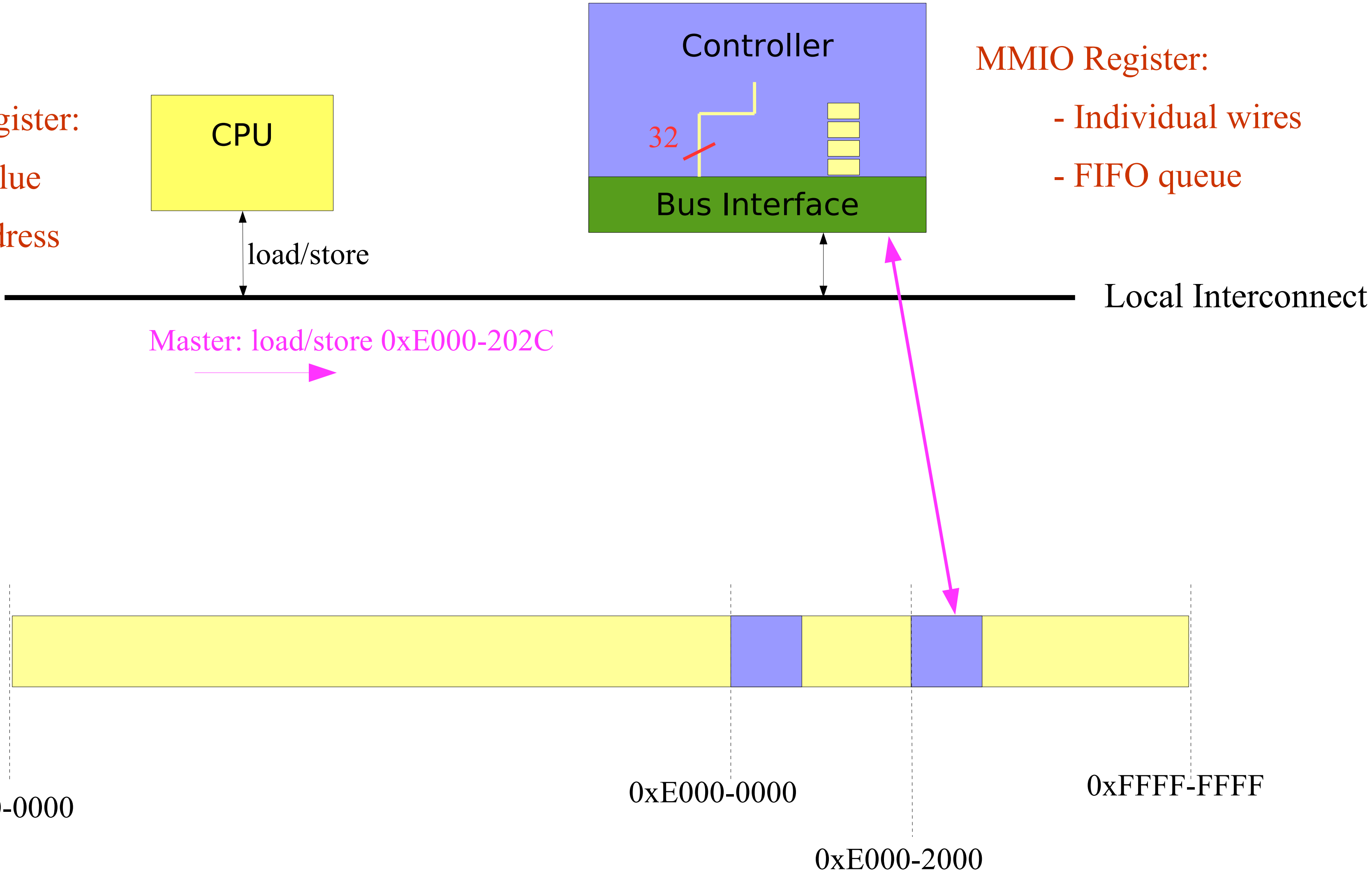
32-bit value

@ an address

MMIO Register:

- Individual wires

- FIFO queue



MMIO: Memory-Mapped Input/Output

Hardware – Zynq-7000 Overview

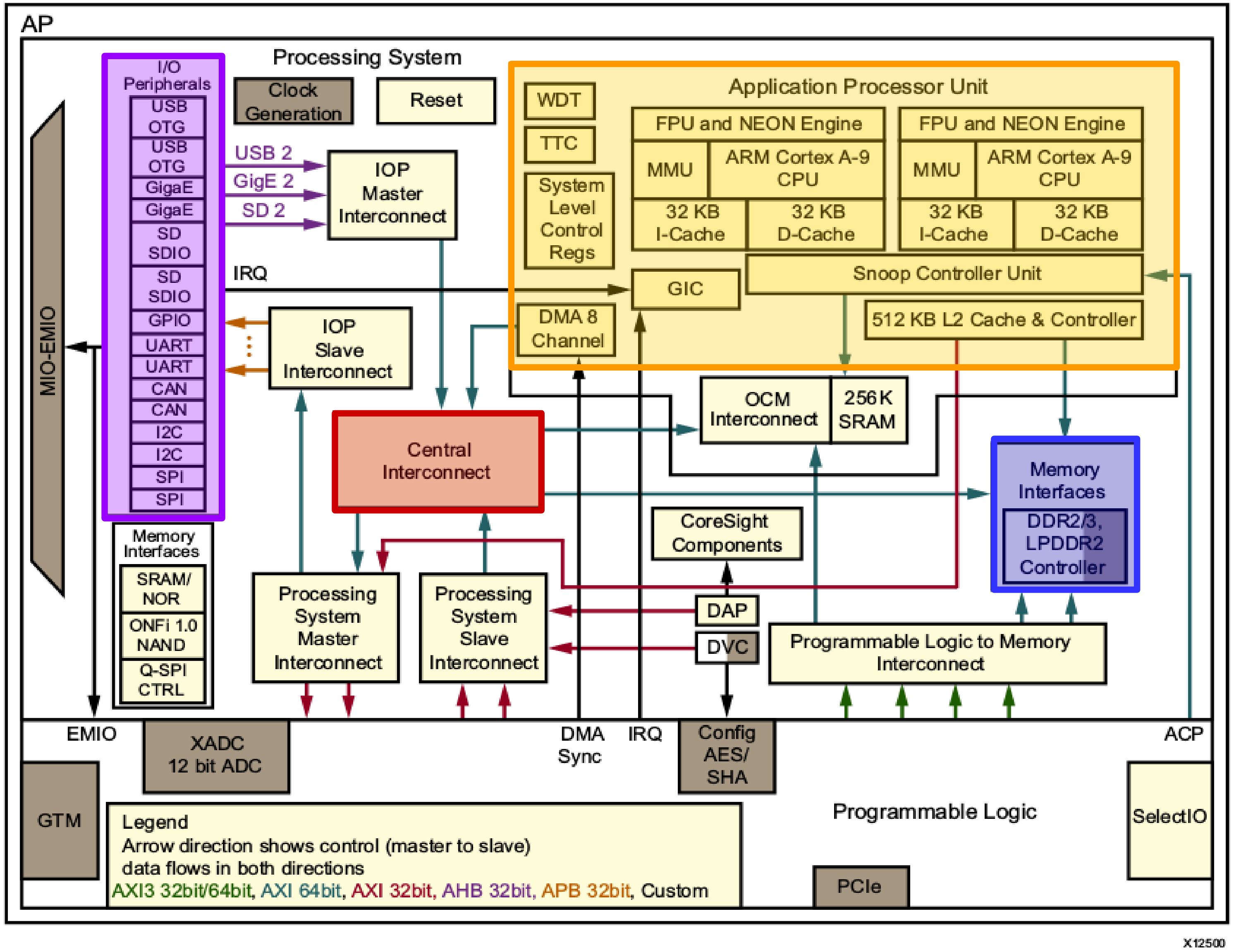


Figure 2-1: **Zynq-7000 AP SoC Processor System High-Level Diagram**

Zynq-7000 Memory Map

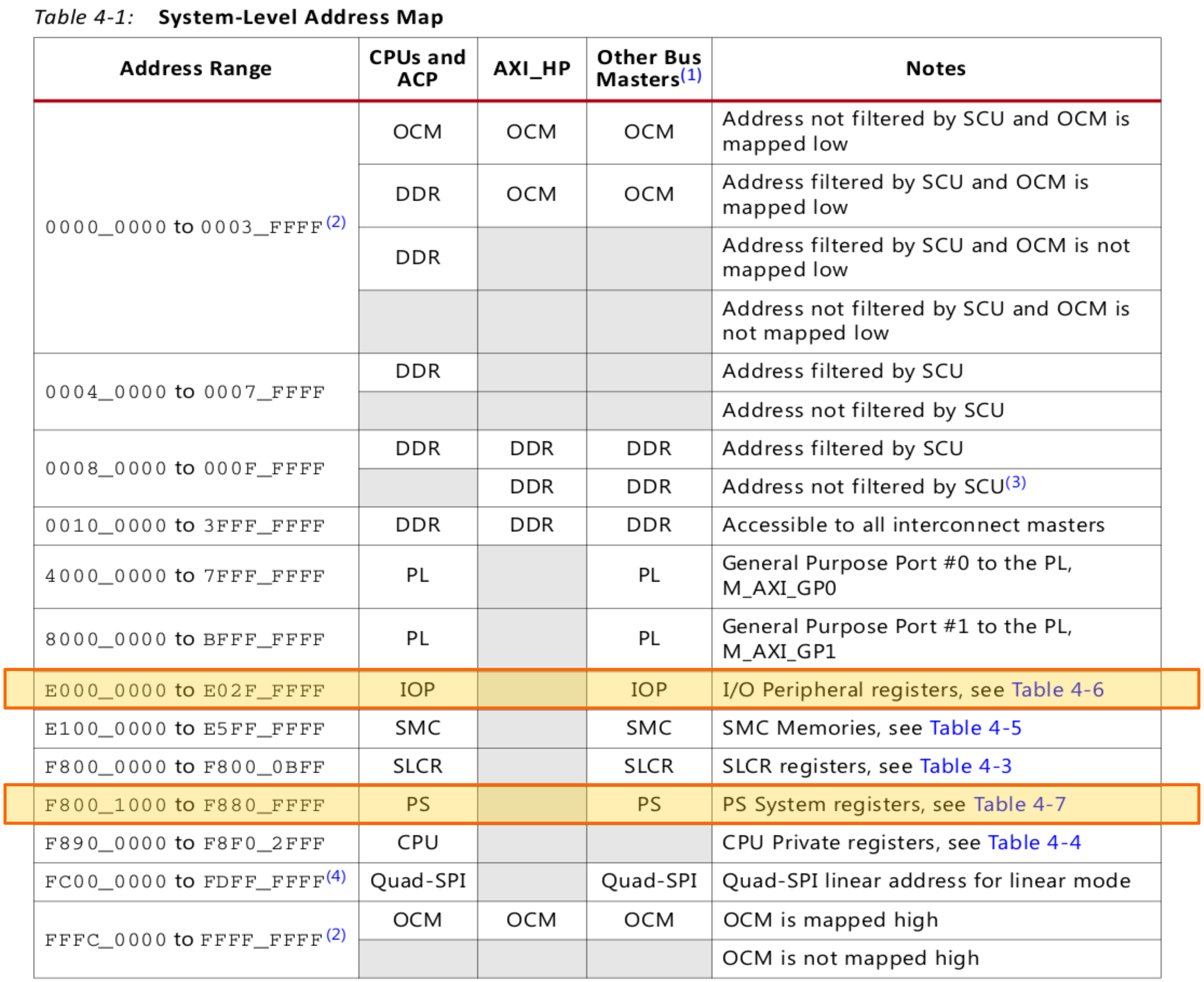


Table 4-1: **System-Level Address Map**

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

Technical Reference Manual: Zynq-7000 All Programmable SoC

Zynq-7000 Memory Map

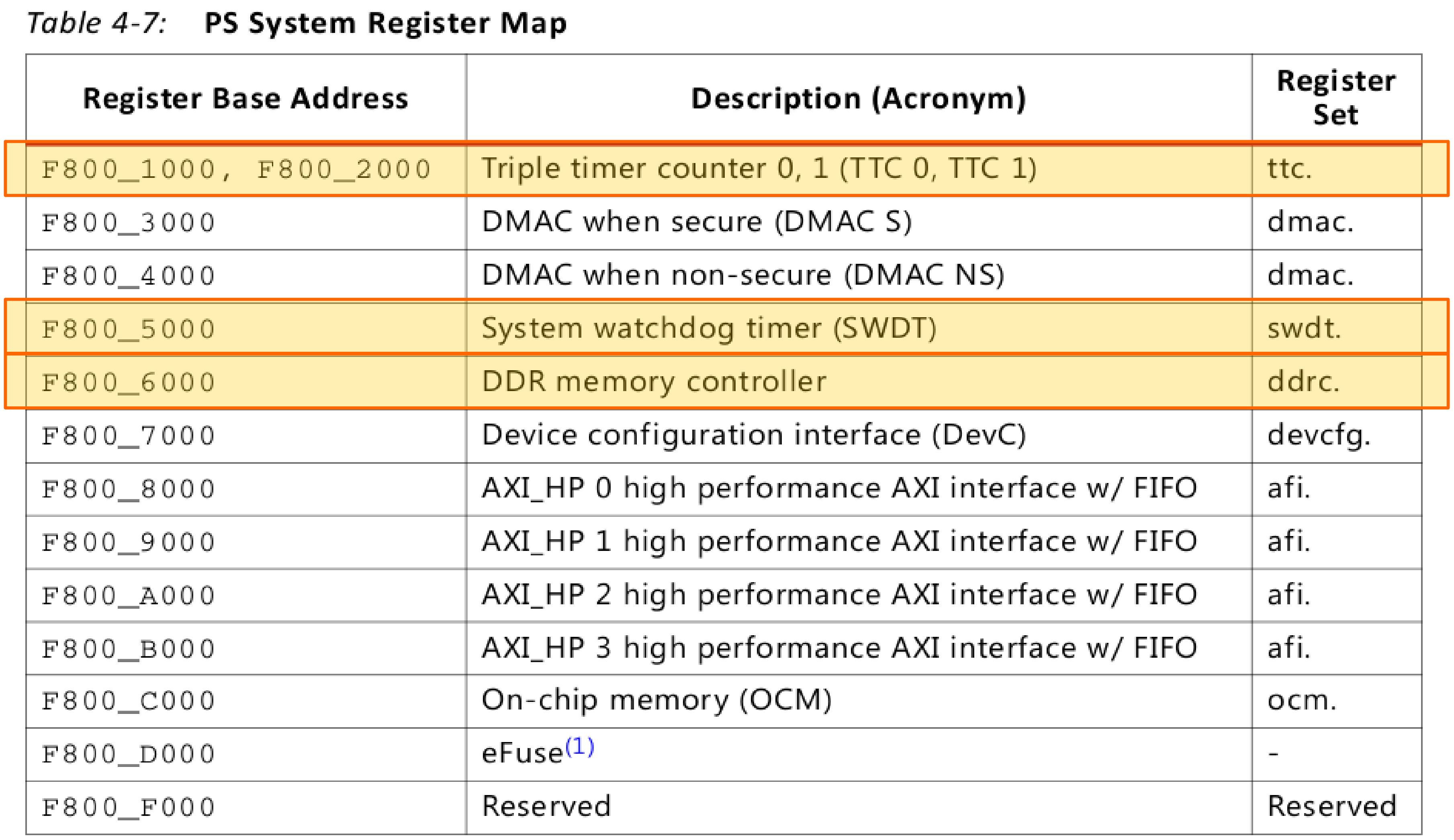


Table 4-7: **PS System Register Map**

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_TEMPTY  (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);

   /* Desactivate 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_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_REMPTY   (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);
}
```

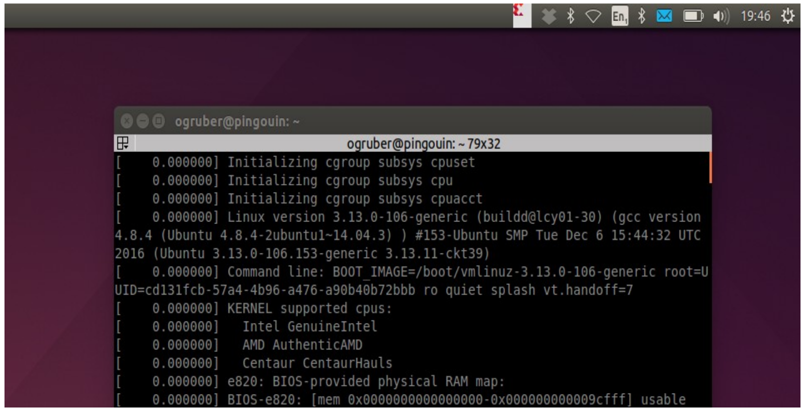
UART – Input

```
#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_REMPTY   (1 <<  1)
#define UART_R1P8_SR_RTRIG    (1 <<  0)
```

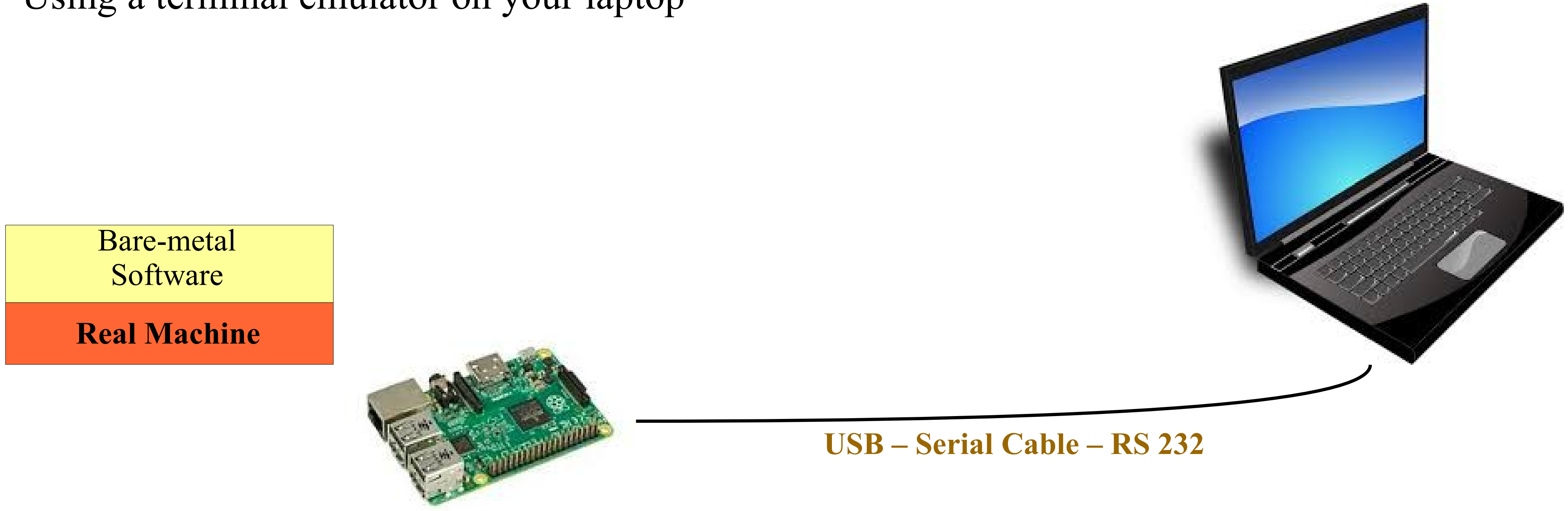
```
uint8_t
uart_r1p8_getc(void* uart){
  while((mmio_read32(uart,UART_R1P8_SR) & UART_R1P8_SR_REMPTY))
    ;
  return mmio_read32(uart,UART_R1P8_FIFO);
}
```

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 the console (a command-line interface)
  - Using a terminal emulator on your laptop

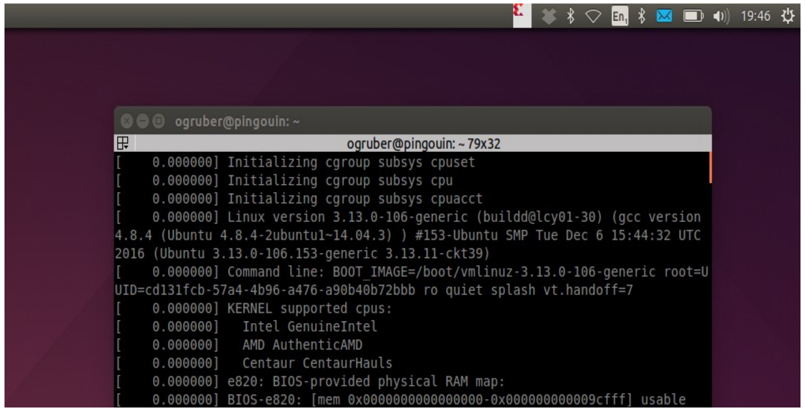


```
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.13.0-106-generic (buildd@lcy01-30) (gcc version
4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.3) ) #153-Ubuntu SMP Tue Dec 6 15:44:32 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=cd131fcb-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
```



Development Environment

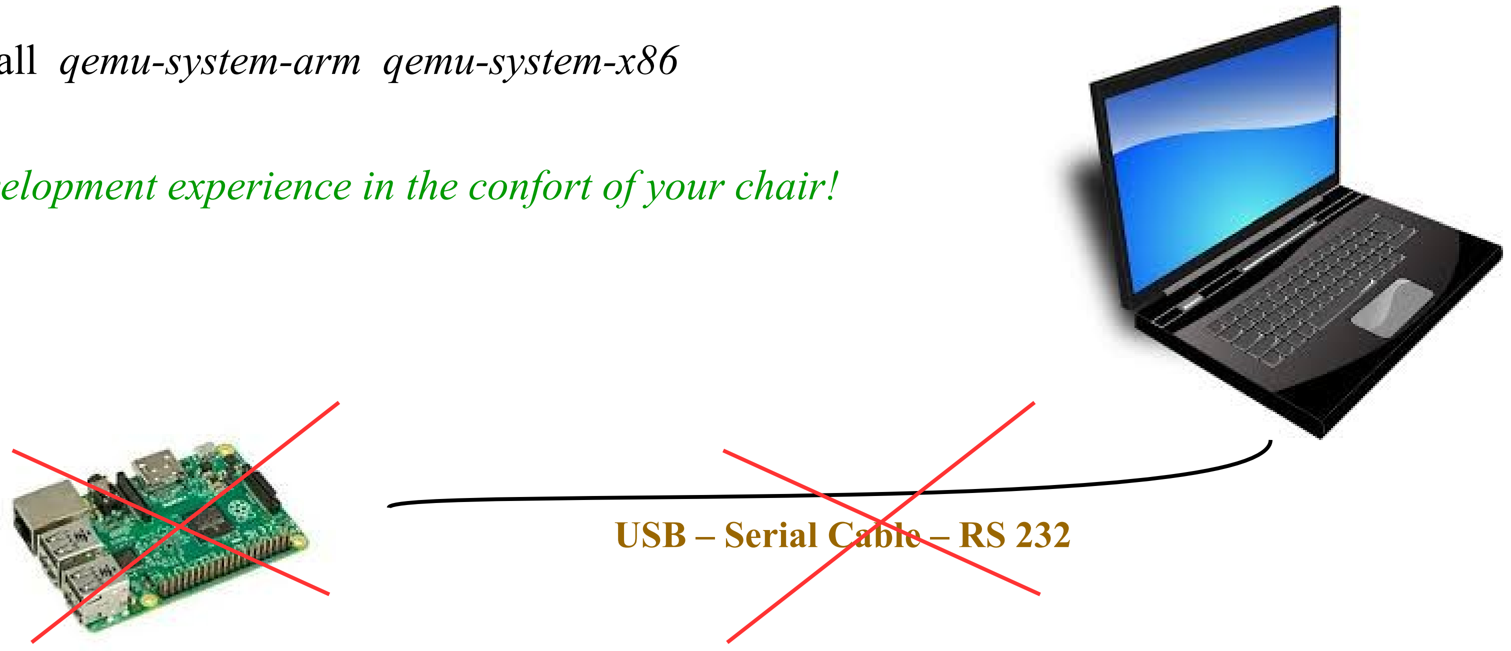
- Using an emulator – QEMU
  - A regular Linux process
  - Emulates a machine for your bare-metal software
  - Direct support for GDB debugging
  - Serial line for a command-line interface



```
[    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 (buildd@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=cd131fcb-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
```

$ sudo apt-get install *qemu-system-arm qemu-system-x86*

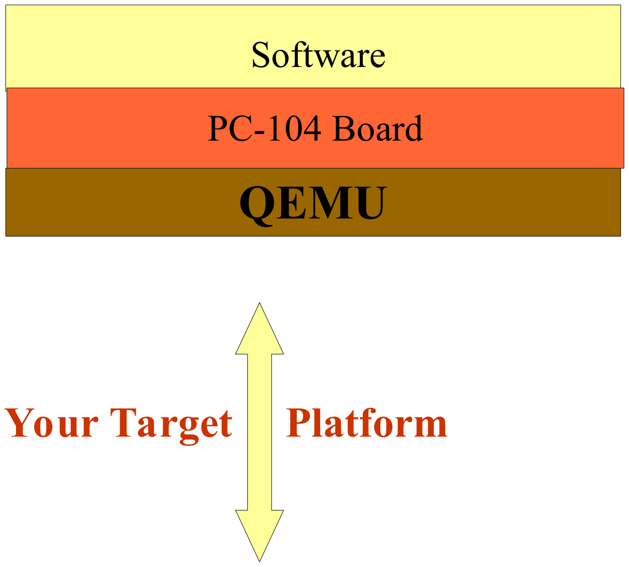
An all-software-development experience in the confort of your chair!

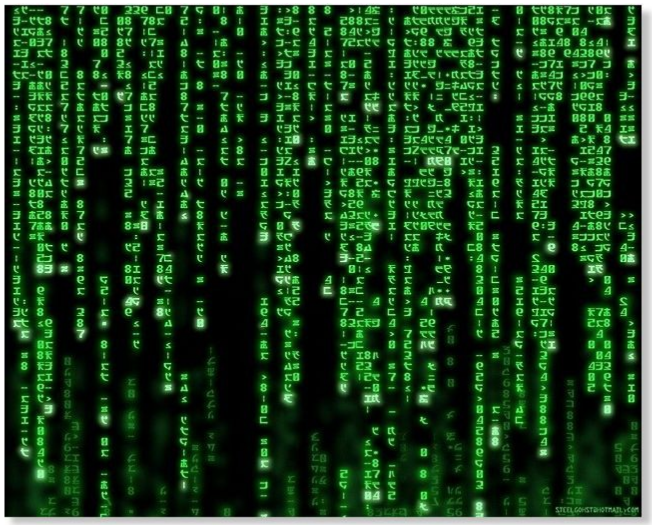


QEMU – Booting an Intel PC

- In-Process Emulator
  - A virtual machine… but looks real to your software…
  - Boots like a regular machine from a virtual disk...
- QEMU is **just** a regular Linux process

```
$ qemu-system-i386 -hda disk.img
```



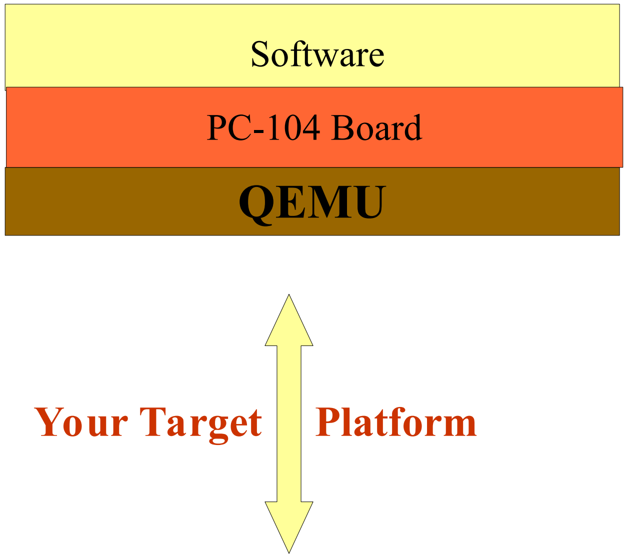


PC-104 Board
→ i386 processor

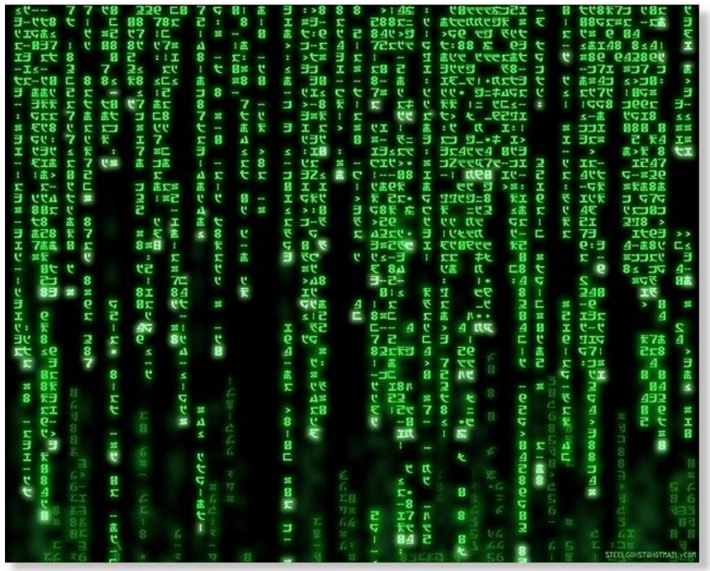
"The matrix is a prison you cannot see, taste, or smell." Morpheus

QEMU – Booting an VersatilePB Board

- Board emulation:
  - VersatilePB board
  - Processor: ARM926EJ-S
- QEMU is **just** a regular Linux process



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



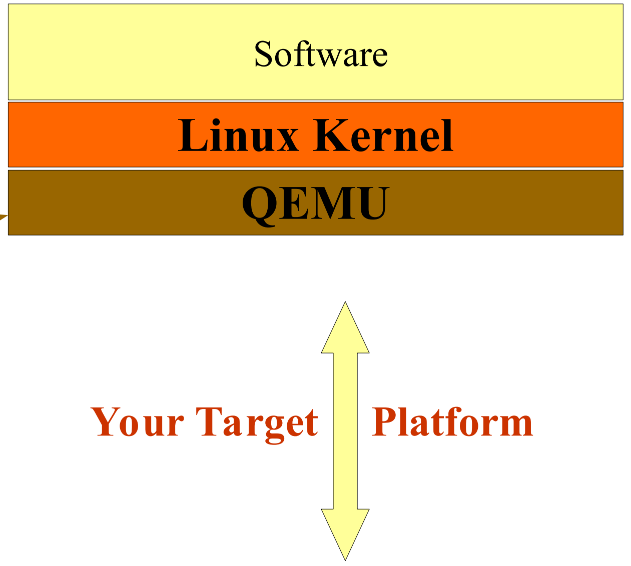
UART0:
Base Address: 0x101F1000

Data register: 0x00
Flag register: 0x18

QEMU – Emulated Hardware

- QEMU Emulated Hardware
  - Information via the 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
```



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...