M2PGi – Internet Of Things

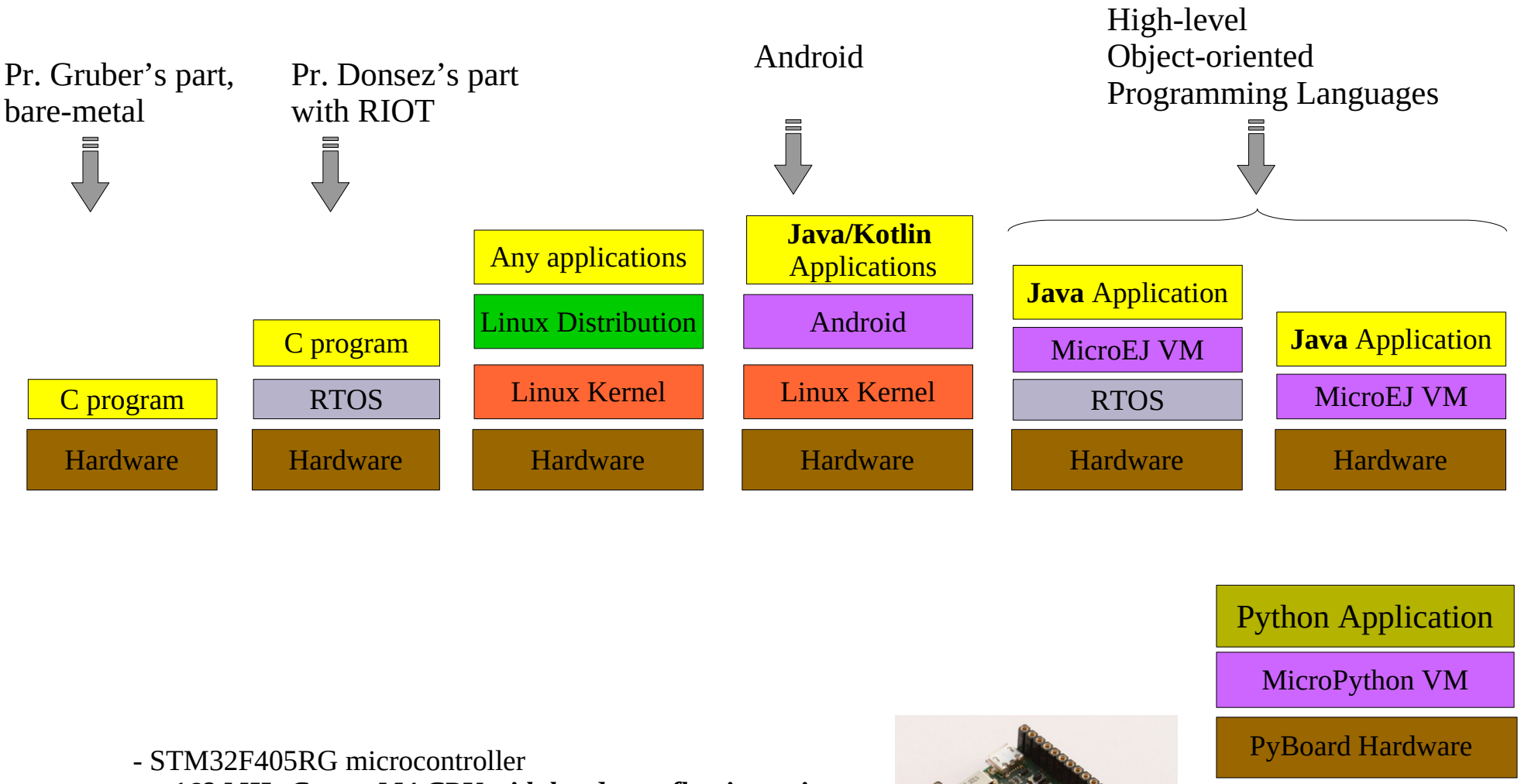
Pr. Didier Donsez (didier.donsez@imag.fr)

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

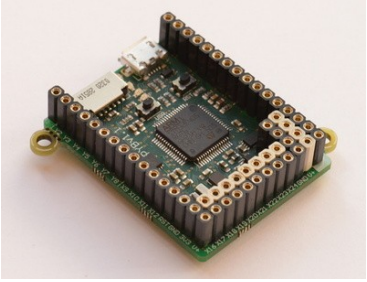
Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Embedded Development Spectrum – Overview

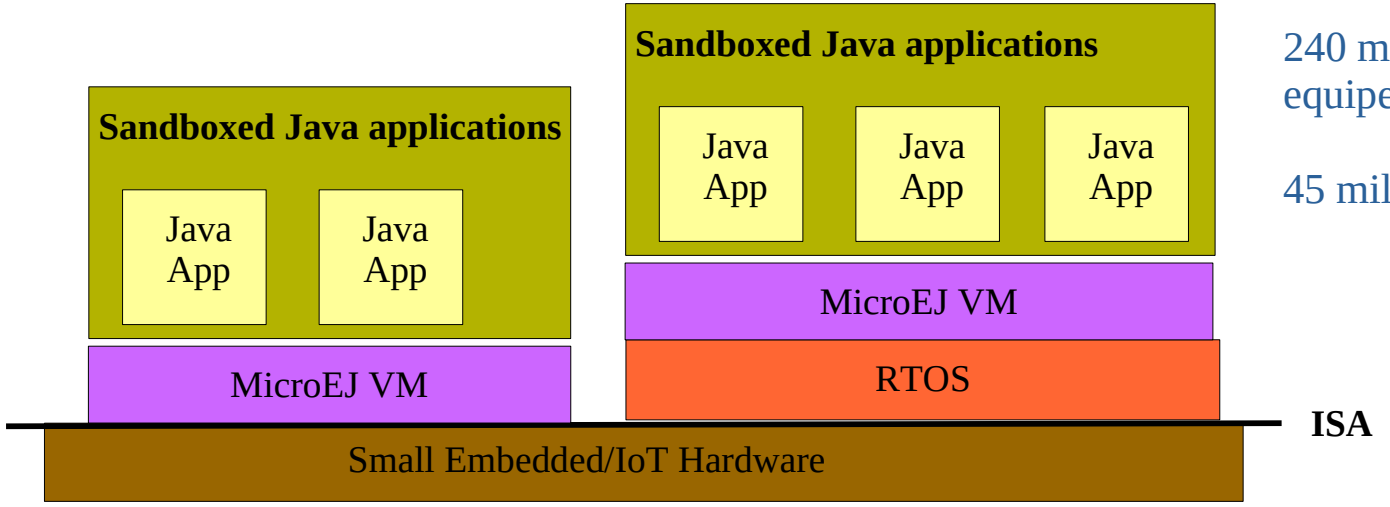


- STM32F405RG microcontroller
  - ***168 MHz Cortex M4 CPU with hardware floating point***
  - **1024 KB flash ROM**
  - **192 KB RAM**
  - Micro USB connector for power and serial communication
  - Micro SD card slot
  - GPIOs



French Tech – MicroEJ[1]

- 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



240 million IoT devices equiped with MicroEJ

45 million euros in R&D

(1) https://www.microej.com/

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...

  - What you probably know less about...

How do we program these?

Which tools do we use?

C program

Hardware

Bare-metal programming

This Part Pedagogy

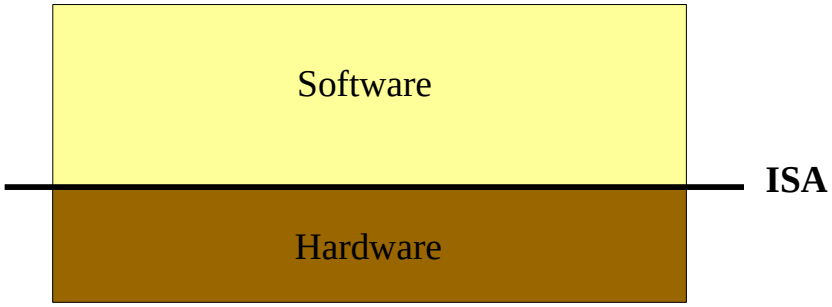
- Team learning, individual work
  - Hands-on learning and coding bare-metal software
  - Your own **work-log:** ***tracking what you do and what you learned***
  - This work-log document is first and foremost for you own records
- The destination is not the goal, the goal is the learning along that path
  - Be curious… expand your skills and know-how
  - Discuss what you learned/understood
  - Explain to others – Ask questions
  - ***WRITE DOWN WHAT YOU HAVE LEARNED***
- Part-I Evaluation
  - No exam – but weekly progress checkpoints
  - ***Every week***: you will surrender your work (***work-log*** and your ***code***)
  - Your weekly involvement is the ***largest part of your final grade***

Bare-metal Development

- Bare-metal Software Basics
  - Runs directly on the "bare metal"
- Instruction Set Architecture (ISA)
  - Defines the instruction set (user and privileged)
  - Defines other privileged concepts (interrupts, traps, page tables, ...)
- Cross-development Toolchain
  - Compiler, linker, debugger, and other tools

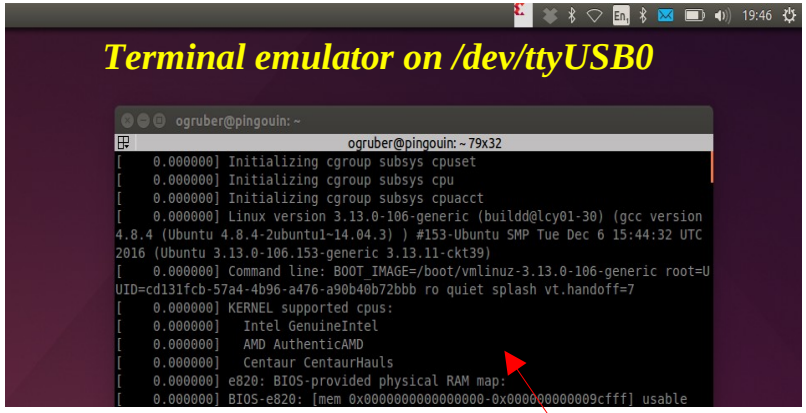
Let's discuss regular Linux development first… with a twist…

It is a warm-up before bare-metal programming



Bare Metal – Development Environment

- Using a real board
  - Relies on using a USB-Serial cable
  - For both JTAG and a console (/dev/ttyUSB0)
- JTAG through /dev/ttyUSB0
  - JTAG to upload the firmware
  - JTAG for hardware and software debugging
- Using a terminal emulator on your laptop
  - Terminal emulator like minicom or kermit
  - ***Serial line for a console*** (a command-line interface)



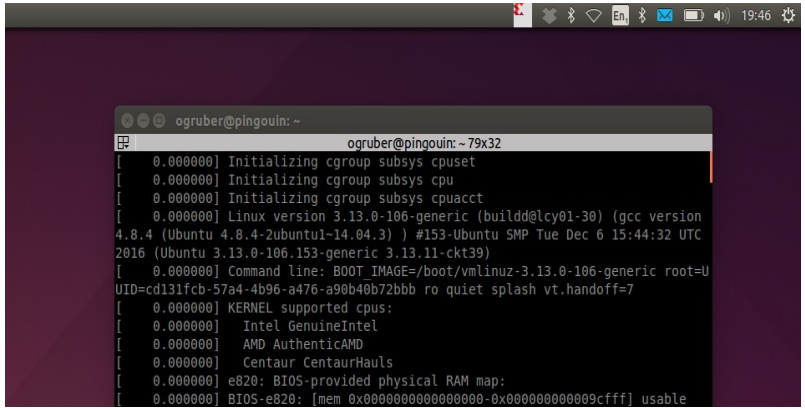
Terminal

Bare-metal Software
Real Machine

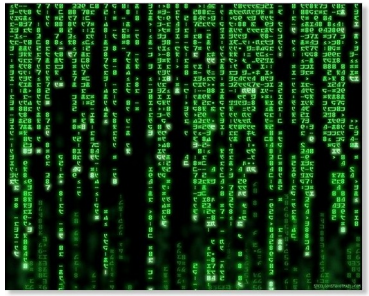
USB – Serial Cable – RS 232
JTAG debugging
Console over serial Line

Bare Metal – 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



For your code: "The matrix is a prison that you cannot see, taste, or smell." Morpheus



Board

Serial Line

Terminal

For you: an all-software-development experience in the confort of your chair!

USB – Serial Cable – RS 232

Cross-compilation & QEMU-based Execution

```
MACHINE=versatilepb
CPU=cortex-a8
MEMSIZE_IN_KB=32

QEMU=qemu-system-arm
QEMU_ARGS= -M $(MACHINE) -cpu $(CPU) -M $(MEMSIZE_IN_KB)K -nographic -serial mon:stdio

CFLAGS= -ggdb -DMEMORY="($(MEMSIZE_IN_KB)*1024)"
ASFLAGS= -ggdb
CFLAGS+= -mcpu=$(CPU) -c -nostdlib -ffreestanding
LDFLAGS= -mcpu=$(CPU) -T kernel.ld -nostdlib -static

# Object files to build and link together
#-----------------------------------------------
OBJS= startup.o main.o exception.o

# Compilation Rules
#-----------------------------------------------
%.o: %.c
     arm-none-eabi-gcc $(CFLAGS) -o $@ $<
%.o: %.s
     arm-none-eabi-as $(ASFLAGS) -o $@ $<

# Build and link all, producing an ELF and binary
#-----------------------------------------------
all: $(OBJS)
     arm-none-eabi-ld $(LDFLAGS) $^ -o kernel.elf
     arm-none-eabi-objcopy -O binary kernel.elf kernel.bin

run: all
  $(QEMU) $(QEMU_ARGS) -device loader,file=kernel.elf

debug: all
  $(QEMU) $(QEMU_ARGS) -device loader,file=kernel.elf -gdb tcp::1234 -S
```

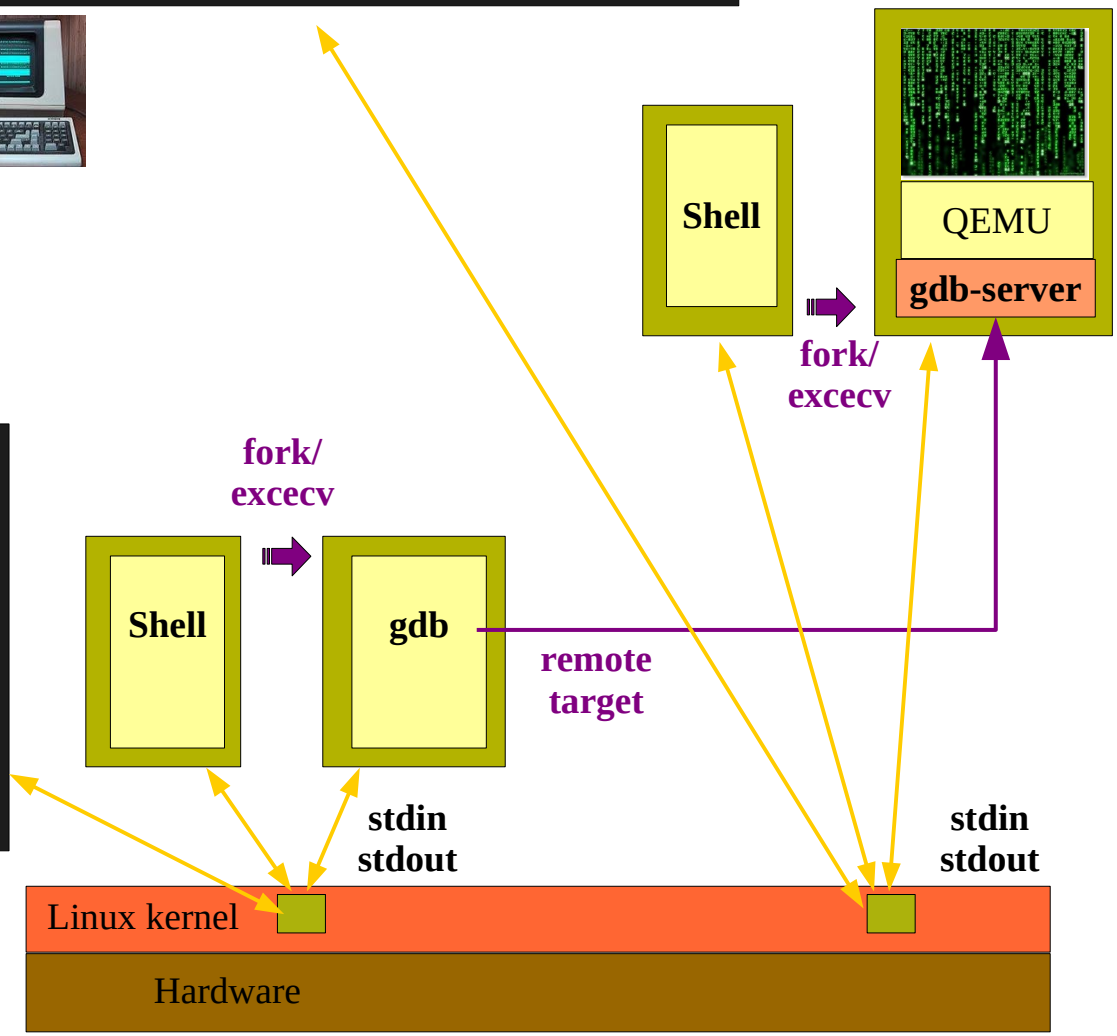
Debugging with QEMU

In one terminal:
- cross-compile and link your kernel
- launch QEMU to load your kernel

```
$ arm-none-eabi-gcc -ggdb -o kernel.elf *.o
$ qemu-system-arm … kernel.elf
```

In another terminal:
- launch your ARM debugger as usual,
- but attach to a remote target

```
$ arm-none-eabi-gdb kernel.elf
…
(gdb) target remote localhost:1234
Remote debugging using :1234
?? () at exception.s:31
31        ldr pc, reset_handler_addr
(gdb)
```



Nota Bene: use short notation
tar rem :1234
target remote localhost:1234

Debugging with QEMU


```
$ qemu-system-arm -M versatilepb -cpu cortex-a8 -m "32K"
  -nographic -serial mon:stdio -device loader,file=kernel.elf
```

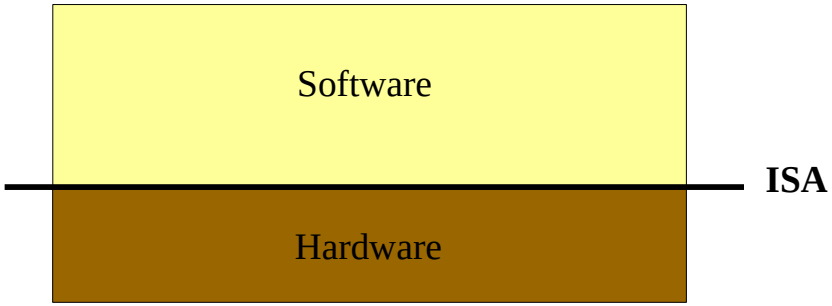
```
$ gdb-multiarch kernel.elf
gdb-multiarch kernel.elf
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 Free Software Foundation, Inc.

(gdb) target remote localhost:1234
Remote debugging using :1234
?? () at exception.s:31
31        ldr pc, reset_handler_addr
(gdb) print /x $r15
$1 = 0x0
(gdb)
```

Where is the execution stopped at?
→ At address 0x0000-0000.

Why is it stopped there?
What code is there to execute?
How did we build that code?

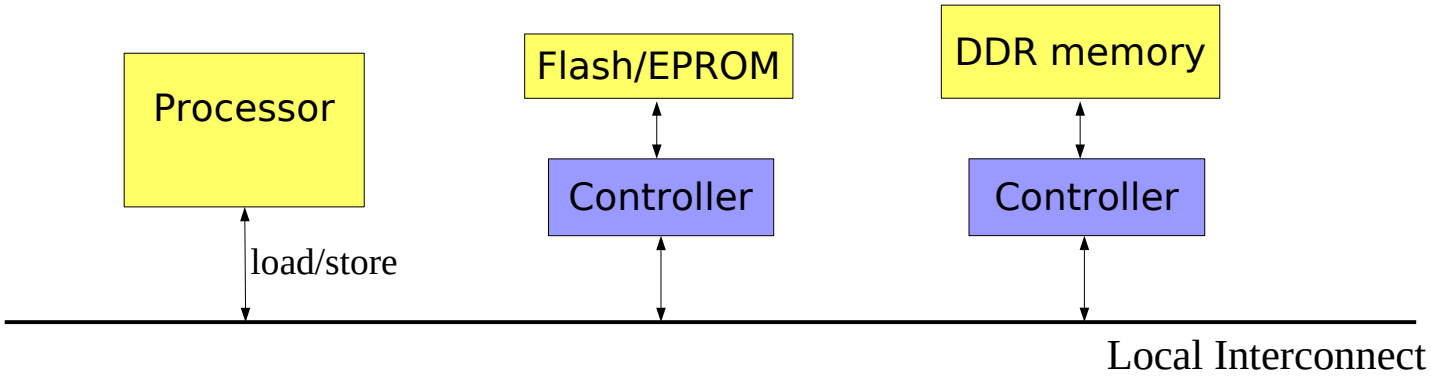
Bare-metal Development Basics

- Bare-metal Software
  - Runs directly on the "bare metal"
- Instruction Set Architecture (ISA)
  - Defines the instruction set (user and privileged)
  - Defines other privileged concepts (page tables, traps, interrupts, ...)
- Cross-development Toolchain
  - Compiler, linker, debugger, and other tools
- Using Integrated Development Environments (IDEs)

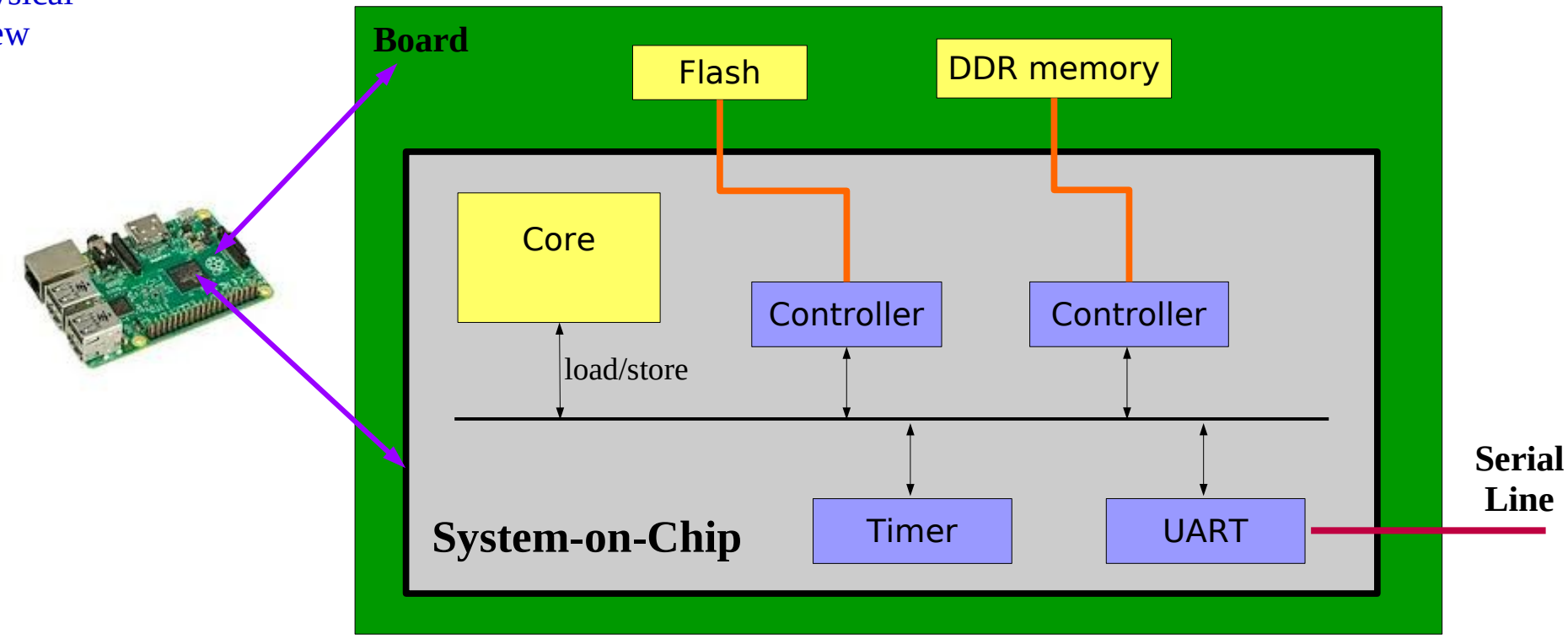


Hardware – Basics

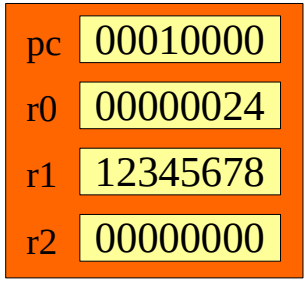
Conceptual View



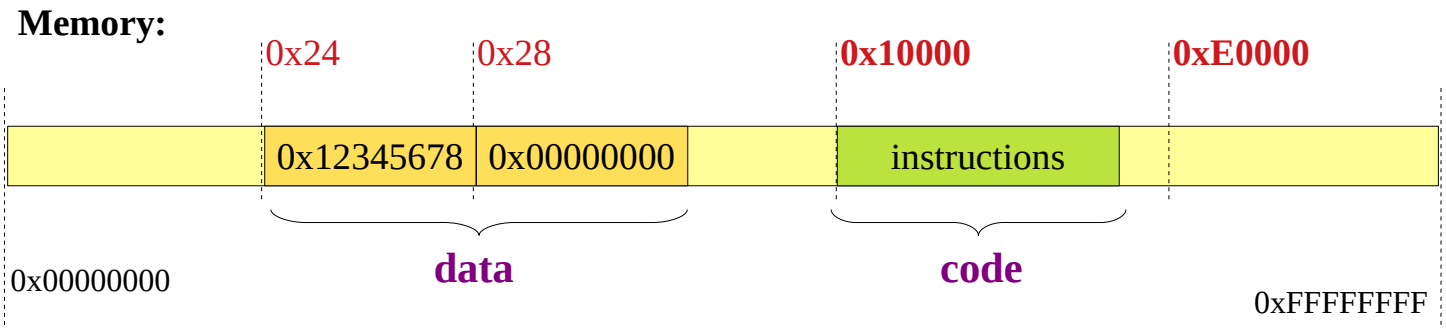
Physical View



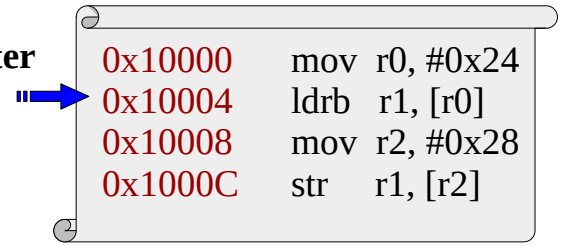
Functional View – Processor & Memory



Registers



Program Counter (pc register)



Code

Processor :

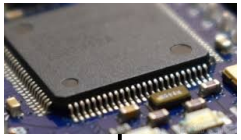
Fetch instruction **@pc**

Decode instruction

pc = pc + sizeof(instruction)

Execute instruction

Processor



Only two operations:
- value ← **load**(address)
- **store**(address,value)

both for data and code...

Memory

Controller

Bus/Interconnect

Zynq-7000 – Boot Sequence & Boot Loader Stage1

Processor Reset
→ pc=0x0 000-0000

With:
OCM is mapped low

Boot Steps:
- initialize DDR controller
- copy code to DDR above 0x0004-0000
- jump there
- map OCM high
- initialize devices
- **Load next stage…**

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

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000-0000 to 0000-FFFF	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
0000_0000 to 0003_FFFF[2]	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 – Boot Stage2 & Normal Memory Map

Memory Map:
Normal memory (DDR)
from 0x0000-00000
to 0x3FFF-FFFF

Processor Reset:
→ pc=0x0 000-0000

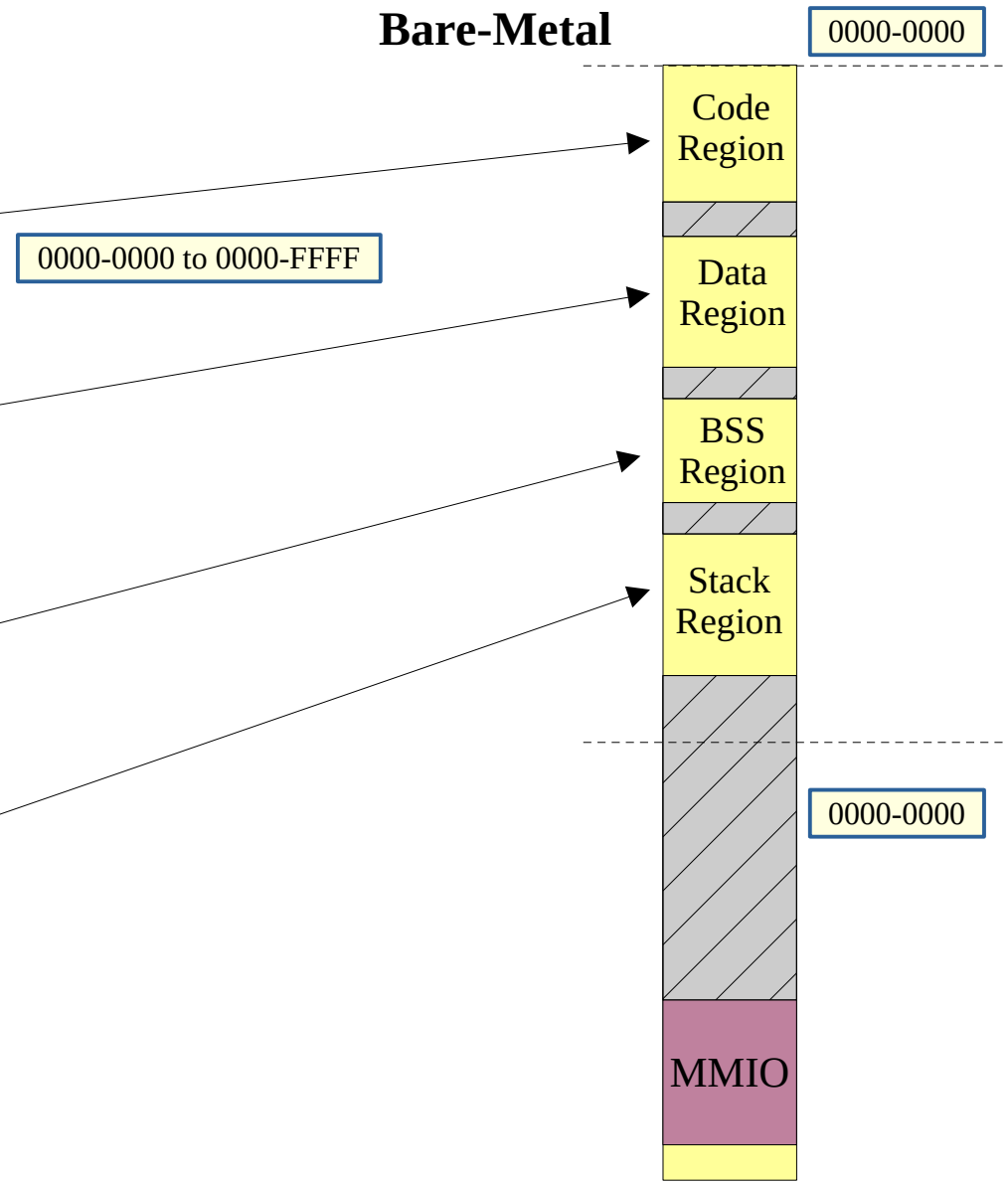
So the entry point of the stage two kernel is at 0x0000-0000

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

Linker Script – Code / Data Memory Layout


```
SECTIONS
{
  . = 0x00;
  .text : {
     build/exception.o(.text)
     build/startup.o(.text)
     build/*(.text)
  }
  . = ALIGN(4);
  .data : {
    build/*(.data)
  }
  . = ALIGN(4);
   _bss_start = .;
  .bss . : {
   build/*(.bss COMMON)
  }
  . = ALIGN(4);
  _bss_end = .;

  . = ALIGN(8);
  . = . + 0x1000;
  stack_top = .;
}
```



Assembly Exception Vector & Code (“exception.s”)

```
// will be loaded at 0x0000-0000
     ldr pc, reset_handler_addr
     ldr pc, undef_handler_addr
     ldr pc, swi_handler_addr
     ldr pc, prefetch_abort_handler_addr
     ldr pc, data_abort_handler_addr
     ldr pc, unused_addr
     ldr pc, irq_handler_addr
     ldr pc, fiq_handler_addr

// see previous slide: assembly boot code...
reset_handler_addr: .word _reset_handler


undef_handler_addr: .word _undef_handler
swi_handler_addr: .word _swi_handler
prefetch_abort_handler_addr: .word _prefetch_abort_handler
data_abort_handler_addr: .word _data_abort_handler
not_used_addr: .word _not_used_handler
irq_handler_addr: .word _isr_handler
fiq_handler_addr: .word _fiq_handler

_isr_handler:
     b _isr_handler
_unused_handler:
     b _unused_handler // unexpected fast interrupt
_fiq_handler:
     b _fiq_handler // unexpected fast interrupt
_undef_handler:
     b _undef_handler // unexpected trap for an undefined instruction
_swi_handler:
     b _swi_handler  // unexpected software interrup
_prefetch_abort_handler:
     b _prefetch_abort_handler  // unexpected prefetch-abort trap
_data_abort_handler:
     b _data_abort_handler  // unexpected abort trap
```

Stage2 execution starts at 0x0000-0000...

This is where QEMU starts the execution, after loading your ELF file. In other words, QEMU provides the state1.

Use different labels and loops to see the reason when something did go wrong...
Especially when accessing invalid Memory via stray pointers...

Cortex-A8 – Assembly Boot Code (“startup.s”)

```
.global _reset_handler
_reset_handler:

   /*
    * Set the core in the SYS_MODE, with all interrupts disabled.
    */
   msr     cpsr_c,#(CPSR_SYS_MODE | CPSR_IRQ_FLAG | CPSR_FIQ_FLAG)

   /* set the C stack pointer for the SYS_MODE */
   ldr     sp,=stack_top

   /*
    * Clear out the bss section, located from _bss_start to _bss_end.
    * This is a C convention, the GNU GCC compiler will group
    * all global variables that need to be zeroed on startup
    * in the bss section of the ELF.
    */
   ldr     r4, =_bss_start
   ldr     r9, =_bss_end
   mov     r5, #0
1:
   stmia   r4!, {r5}
   cmp     r4, r9
   blo     1b

   /*
    * Set the GCC frame-pointer to null
    * and then upcall the C entry function  _start(void)
    */
   eor r11, r11, r11
   ldr r3,=_start
   blx r3
halt:
    b halt  // in case the function "_start" returns...
```

Setup the processor mode, and disable all interrupts Then set the stack pointer.

Then clear the "bss data"

Done with assembly, for now, upcalling C code...

Processor Modes – ARM Processor

Privileged modes

Exception modes

User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

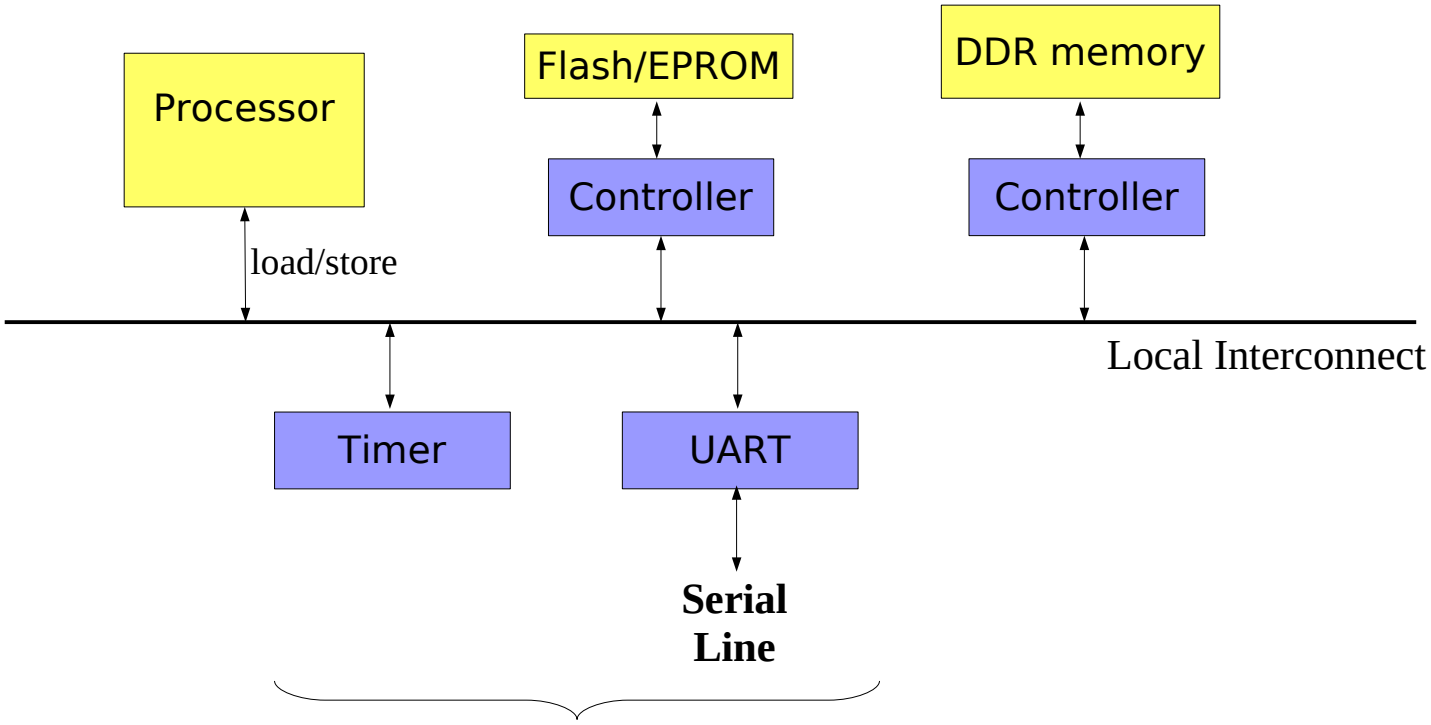
Banked registers

Banked register

indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

We will discuss interrupts later...

Hardware – Peripherals



Without peripherals,
a system can't do much...

Terminal

Serial Line

VersatilePB

Zynq-7000 Memory Map – **Peripherals**

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

Range of memory
to dialog with peripherals

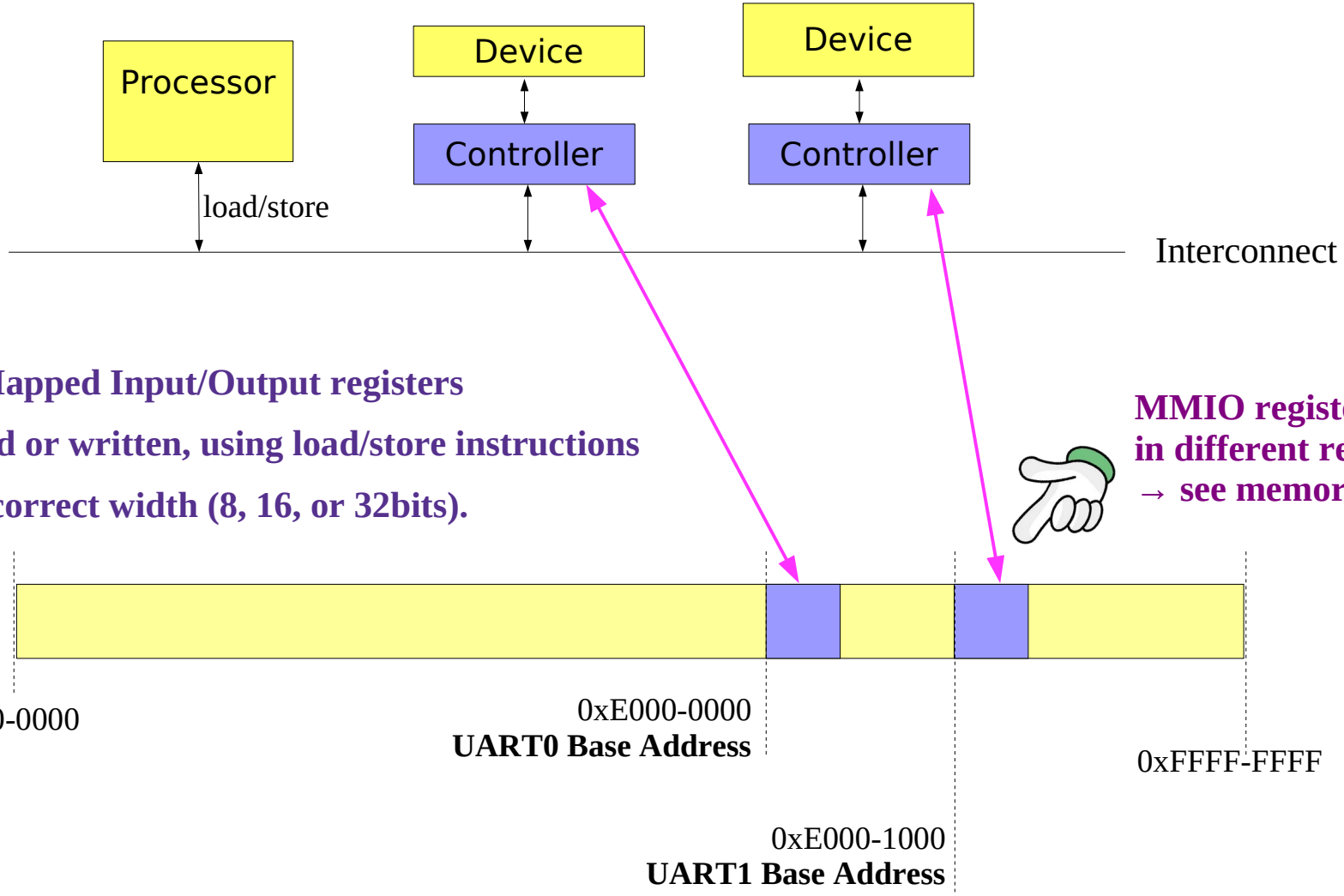
Zynq-7000 Memory Map

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

Technical Reference Manual: Zynq-7000 All Programmable SoC

Memory-Mapped Input/Output Registers (MMIO)



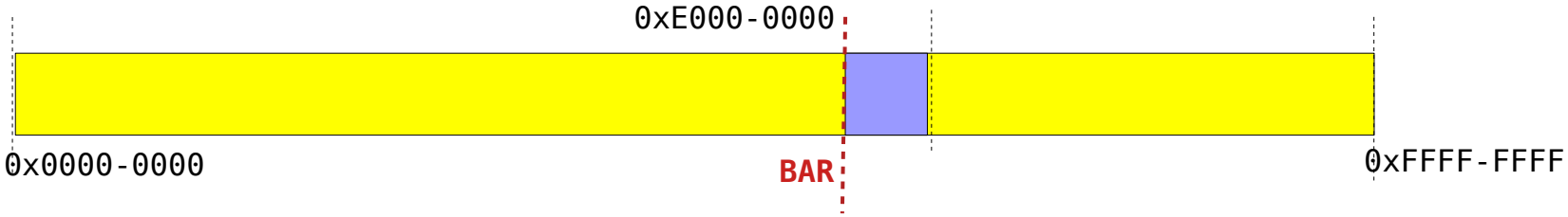
Memory-Mapped Input/Output registers

Can be read or written, using load/store instructions

But of the correct width (8, 16, or 32bits).

MMIO registers in different regions → see memory map

Manipulating MMIO registers




```c
__inline__
__attribute__((always_inline))
uint16_t mmio_read16(void* bar, uint32_t offset) {
  return *((uint16_t*)(bar+offset));
}

__inline__
__attribute__((always_inline))
void mmio_write16(void* bar, uint32_t offset, uint16_t value) {
  *((uint16_t*)(bar+offset)) = value;
}

__inline__
__attribute__((always_inline))
void mmio_setbits(void* bar, uint32_t offset, uint32_t bits) {
  uint32_t value = *((uint32_t*)(bar+offset));
  value |= bits;
  *((uint32_t*)(bar+offset)) = value;
}

__inline__
__attribute__((always_inline))
void mmio_clearbits(void* bar, uint32_t offset, uint32_t bits) {
  uint32_t value = *((uint32_t*)(bar+offset));
  value &= ~bits;
  *((uint32_t*)(bar+offset)) = value;
}
```

would need all the different widths: 8, 16, 32

set/clear some bits, preserving the others,

nota bene: might be used for either **set** or **clear registers**.

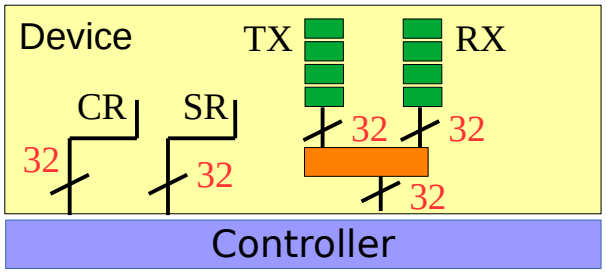
UART Device Example

From the Zynq-7000/R1P8 Technical Reference Manual

- Control Register (32-bits ***CR***)
- Status Register (32-bits ***SR***)
- Transmit (***TX***) & Receive (***RX***) FIFO

relevant mmio registers

FIFO depths: 8 registers




```
#define UART_CR          0x0000 /* UART Control Register */
#define UART_MR          0x0004 /* UART Mode Register */
#define UART_IER         0x0008 /* -- Interrupt Enable Register */
#define UART_IDR         0x000C /* -- Interrupt Disable Register */
#define UART_IMR         0x0010 /* -- Interrupt Mask Register */
#define UART_ISR         0x0014 /* -- Channel Interrupt Status Register */
#define UART_BAUDGEN     0x0018 /* Baude Rate Generator Register */
#define UART_RXTOUT      0x001C /* -- Receiver Timeout Register */
#define UART_RXWM        0x0020 /* -- Receiver FIFO Trigger Level Register */
#define UART_MODEMCR     0x0024 /* -- Modem Control Register */
#define UART_MODEMSR     0x0028 /* -- Modem Status Register */
#define UART_SR          0x002C /* Channel Status Register */
#define UART_FIFO        0x0030 /* Transmit & Receive FIFO */
#define UART_BAUDDIV     0x0034 /* Baud Rate Divider Register */
#define UART_FLOWD       0x0038 /* -- Flow Control Delay Register */
#define UART_TXWM        0x0044 /* -- Transmitter FIFO Trigger Level Register */
```

FIFO Queues
TX RX

UART – Input


```
#define UART_SR          0x002C /* Channel Status Register */
#define UART_FIFO        0x0030 /* Transmit & Receive FIFO */
/*
 * Channel Status Register (UART_SR)
 * Bit-field masks:
 */
#define UART_SR_TNFUL    (1 << 14)
#define UART_SR_TTRIG    (1 << 13)
#define UART_SR_FDELT    (1 << 12)
#define UART_SR_TACTIVE  (1 << 11)
#define UART_SR_RACTIVE  (1 << 10)
#define UART_SR_TFUL     (1 <<  4)
#define UART_SR_TEMPTY   (1 <<  3)
#define UART_SR_RFUL     (1 <<  2)
#define UART_SR_REMPTY   (1 <<  1)    // Mask to know if the RX FIFO is empty.
#define UART_SR_RTRIG    (1 <<  0)
```

```
/*
 * Reads one byte, if available, otherwise spin-waits.
 */
uint8_t
uart_read(void* uart){
  while((mmio_read32(uart,UART_SR) & UART_SR_REMPTY))
    ;
  return mmio_read32(uart,UART_FIFO);
}
```

UART – Input


```
#define UART_SR          0x002C /* Channel Status Register */
#define UART_FIFO        0x0030 /* Transmit & Receive FIFO */
/*
 * Channel Status Register (UART_SR)
 * Bit-field masks:
 */
#define UART_SR_TNFUL    (1 << 14)
#define UART_SR_TTRIG    (1 << 13)
#define UART_SR_FDELT    (1 << 12)
#define UART_SR_TACTIVE  (1 << 11)
#define UART_SR_RACTIVE  (1 << 10)
#define UART_SR_TFUL     (1 << 4)    // Mask to know if the TX FIFO is full.
#define UART_SR_TEMPTY   (1 << 3)
#define UART_SR_RFUL     (1 << 2)
#define UART_SR_REMPTY   (1 << 1)
#define UART_SR_RTRIG    (1 << 0)
```

```
/*
 * Writes one byte, if there is room to do so, otherwise spin-waits.
 */
void
uart_write(void* uart, uint8_t bits){
  while((mmio_read32(uart,UART_SR) & UART_SR_TFUL))
    ;
  mmio_write32(uart,UART_FIFO, bits);
}
```

Our Board – Versatile PB

- Documents
  - Versatile Application Baseboard for ARM926EJ-S User Guide
  - Programmer's Reference, section 4.1, « Memory Map » (page 140)
  - Look for the UART-0 interface base address : 0x101F1000 – 0x101F1FFF
- But how do we know which UART ?
  - Same document, section 3.18 in the hardware description now (page 124)
  - In the Figure 3-31, UARTS are PL011 PrimeCell
- We can also ask QEMU
  - Just to make sure QEMU emulates faithfully the VersatilePB board

4.1 Memory map

The locations for memory, peripherals, and controllers are listed in Table 4-1.

Table 4-1 Memory map

Peripheral	Location	Interrupt[a] PIC and SIC	Address	Region size
MPMC Chip Select 0. Normally the bottom 64MB of SDRAM (During boot remapping, this can be NOR flash, Disk-on-Chip, or static expansion memory)	Board	PIC21, SIC21	0x00000000–0x03FFFFFF	64MB

Table 4-1 Memory map (continued)

Peripheral	Location	Interrupt[a] PIC and SIC	Address	Region size
Smart Card 0 Interface	Dev. chip	PIC 15	0x101F0000–0x101F0FFF	4KB
UART 0 Interface	Dev. chip	PIC 12	0x101F1000–0x101F1FFF	4KB
UART 1 Interface	Dev. chip	PIC 13	0x101F2000–0x101F2FFF	4KB
UART 2 Interface	Dev. chip	PIC 14	0x101F3000–0x101F3FFF	4KB

VersatileAB/PB User Guide

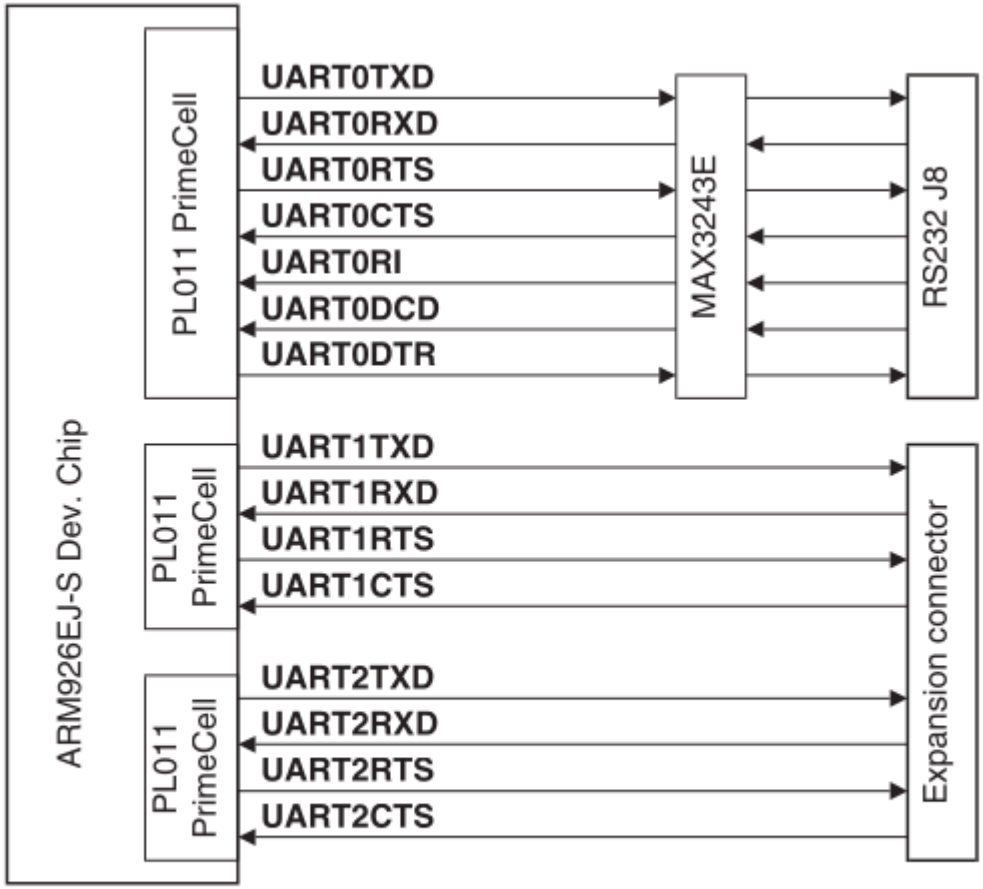


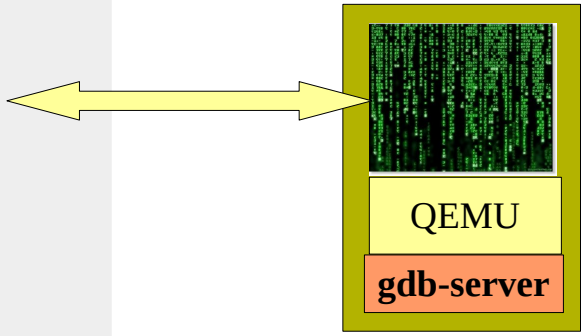
Figure 3-31 UARTs block diagram

Asking QEMU – What is the Illusion?

Open the QEMU console, hit: Ctrl-A c

```
$ qemu-system-arm -M versatilepb -cpu cortex-a8 -m "32K"
  -nographic -serial mon:stdio -device loader,file=kernel.elf

QEMU 8.2.2 monitor - type 'help' for more information
...
(qemu) info qtree
bus: main-system-bus
…
 dev: pl011, id ""
   gpio-out "sysbus-irq" 6
   clock-in "clk" freq_hz=0 Hz
   chardev = "serial0"
   migrate-clk = true
   mmio 00000000101f1000/0000000000001000
…
(qemu)
...
```

Close the QEMU console, hit again: Crtl-A c



To kill QEMU hit: Crtl-A x

Terminal

Serial Line

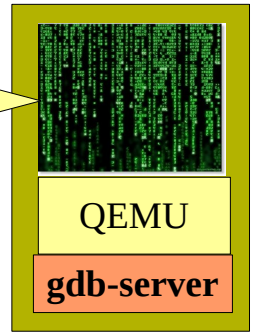
VersatilePB

Asking QEMU – Stop the Illusion!

Open the QEMU console and quit the emulation

Crtl-A c

```
$ qemu-system-arm -M versatilepb -cpu cortex-a8 -m "32K"
  -nographic -serial mon:stdio -device loader,file=kernel.elf

QEMU 8.2.2 monitor - type 'help' for more information
...
(qemu) quit
$
```



Or simply use **Ctrl-A x**

Terminal

Serial Line

VersatilePB

Being Efficient… Master Tools

- QEMU
  - Just enough...
- GNU Toolchain
  - Make / Compiler / Debugger
  - There is no way around it...
- IDE
  - Like VsCode or Eclipse
  - These tools can help your productivity
  - But only if you master them

You really have to know more than just the basics for bare-metal sofware development…

During the first weeks, you will have no other way to debug, there is no "***printf***"...

VsCode – QEMU Debug Configuration

Install plugin: Native Debug from WebFreak

Launch VsCode in the directory where there is your makefile
(this is a single workspace setting)

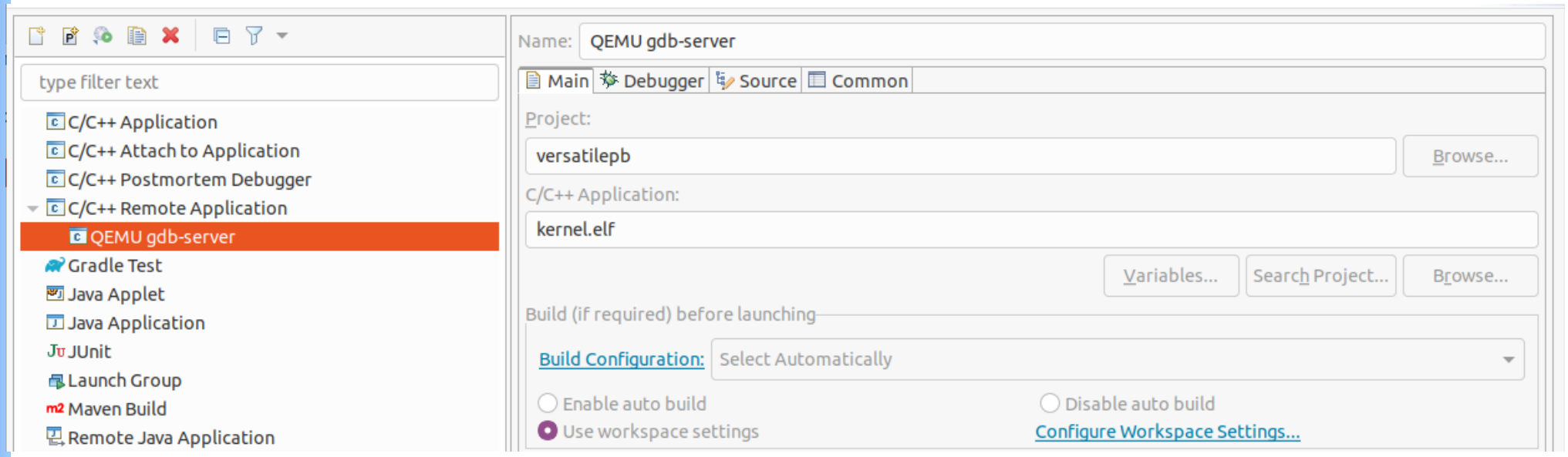
Create **".vscode/launch.json"**
and the configuration to debug by
attaching to the ***QEMU gdb-server***
accepting connections on ***port 1234***.

This configuration assumes the makefile
produced the final executable in
the file "kernel.elf" as an ELF format.

```
{
 "version": "0.2.0",
 "configurations": [
   {
     "name": "Native Debug",
     "type": "gdb",
     "request": "attach",
     "stopAtConnect": true,
     "executable": "${workspaceFolder}/kernel.elf",
     "target": "localhost:1234",
     "remote": true,
     "gdbpath": "gdb-multiarch",
     // "gdbpath": "arm-none-eabi-gdb",
     "cwd": "${workspaceRoot}",
     "autorun": [
        "set substitute-path /vagrant ."
     ]
   }
 ]
}
```

Note: you could use Microsoft pluging for C/C++
but it has a poor support for the GDB console

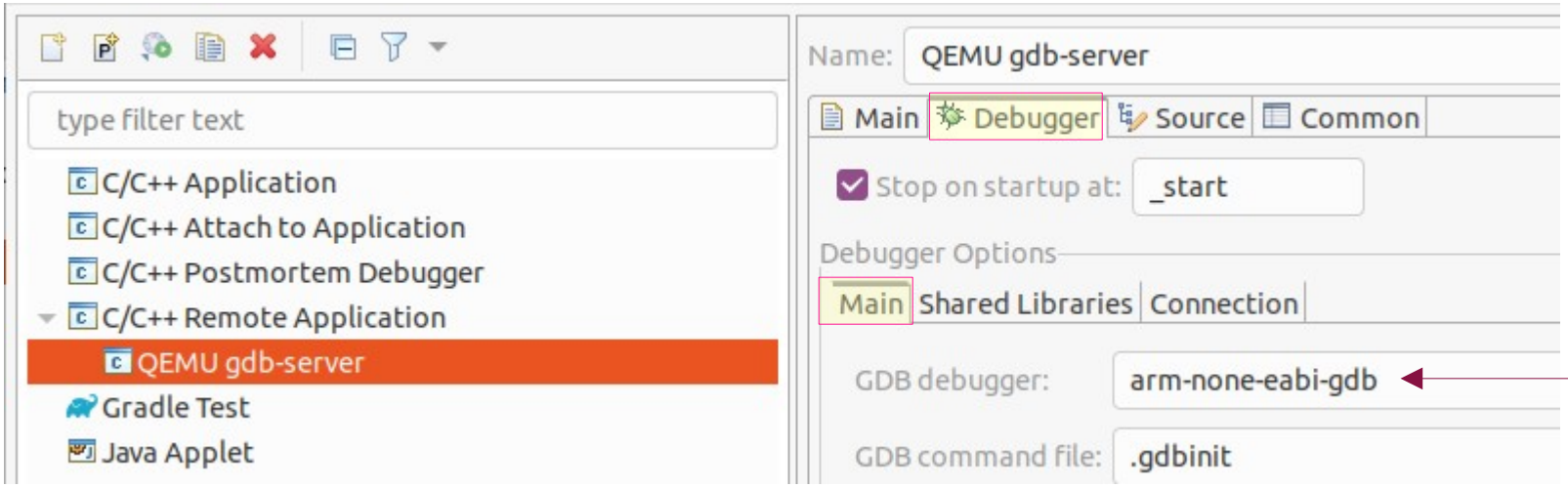
Eclipse – QEMU Debug Configuration



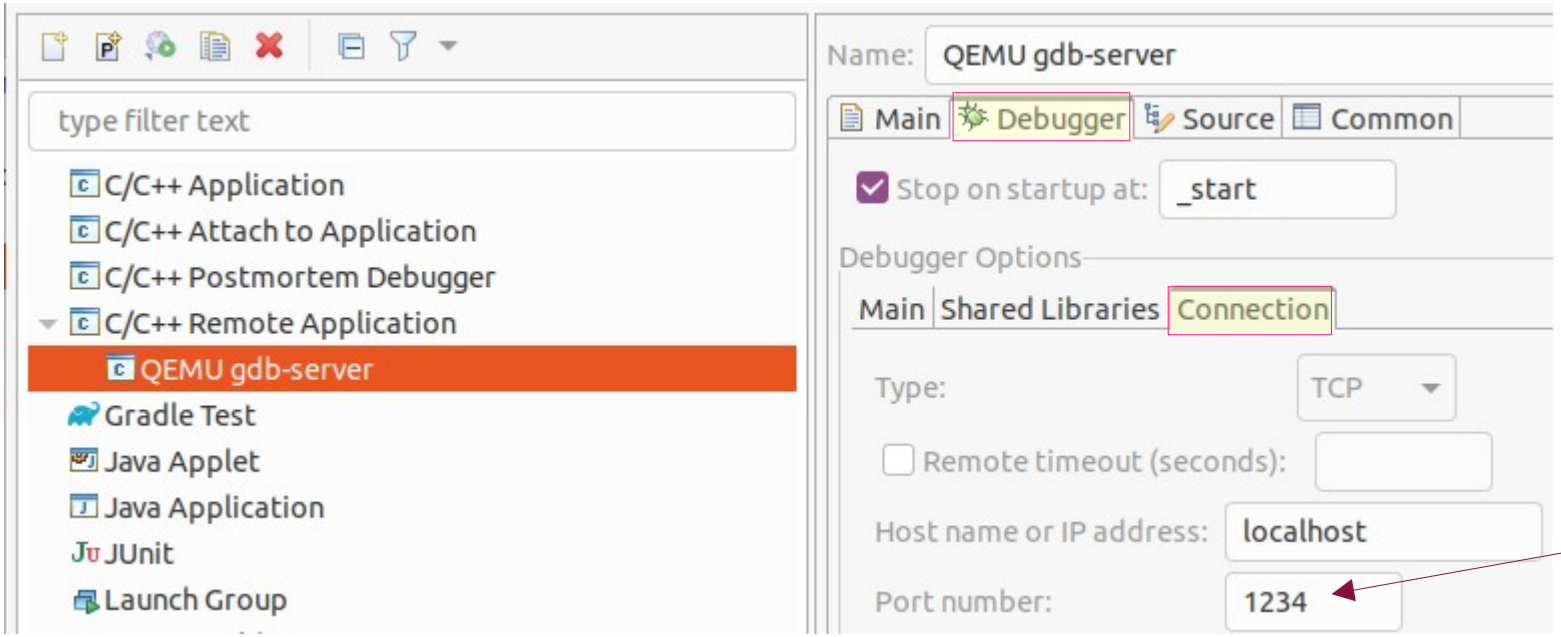
Requires to install the plugins for development in C

Look for the plugin "CDT"

Eclipse – QEMU Debug Configuration



gdb-multiarch



same port as the one given to QEMU

This Week – Expected Work

- Setup
  - Install all the needed software
- Build and run
  - The system will do nothing…
  - It will actually “panic” silently...
- Understand
  - The given project
  - Makefile and source code
- Bare-metal Coding
  - Get the UART0 to work
  - Should provide an “echo console”

There is a lot to understand...

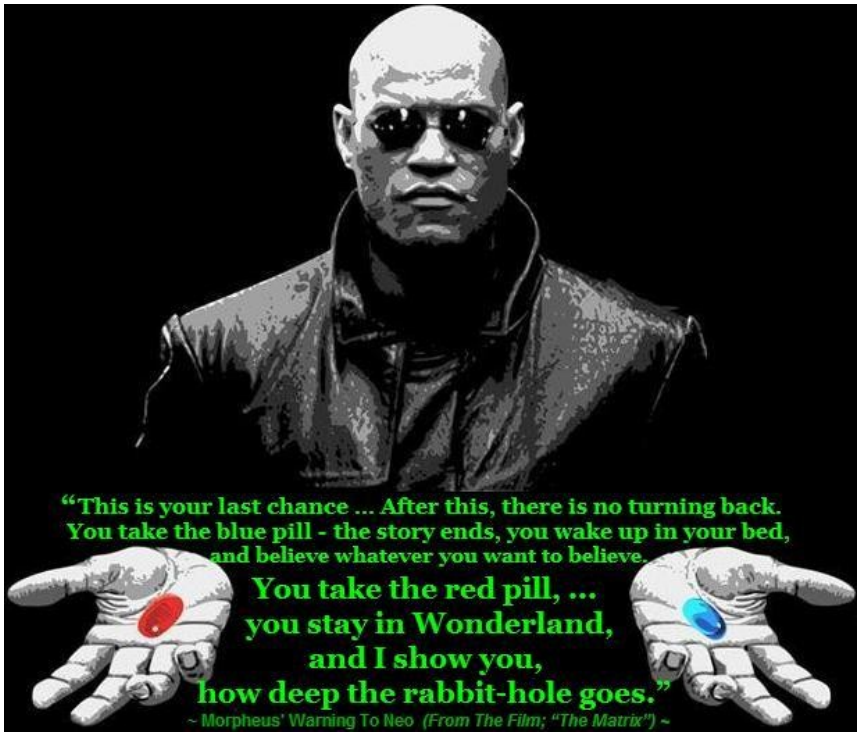
A lot of know-how to acquire…

This means you should have

a lot of questions next week...

Warning...

- The usal ***misunderstanding***...
  - Just reading is not enough
  - Understanding is not know-how
  - Rigorous learning is required
- Work log…
  - Think:
    - What will I need in ***12 months*** to be able to restart on this project with a minimal ramp-up time… About the code and the concepts!
  - Write:
    - In your own words
    - Not copying my slides, or Wikipedia, or ChatGPT…



Evaluation…

- Mostly your work log
  - ***Filled every week, as you do the work***
  - ***Date everything, Never delete anything***
  - But improve, rewrite to clarify/correct
- Examples of sections on what you learn
  - Cross-compiling
  - Compiling and linking for bare-metal
  - Linker script and ELF basics
  - Makefile basics
  - QEMU basics
  - GDB cheat sheet
  - ARM assembly cheat sheet
  - UART PL011 necessary details
- Use a git repository
  - One directory for the worklog
  - One directory per steps
- Over the 4 weeks
  - There will be several steps
  - Use a different git-branch per step
  - Step branches merged back onto the master