Introducing Interrupts

Pr. Olivier Gruber
olivier.gruber@univ-grenoble-alpes.fr

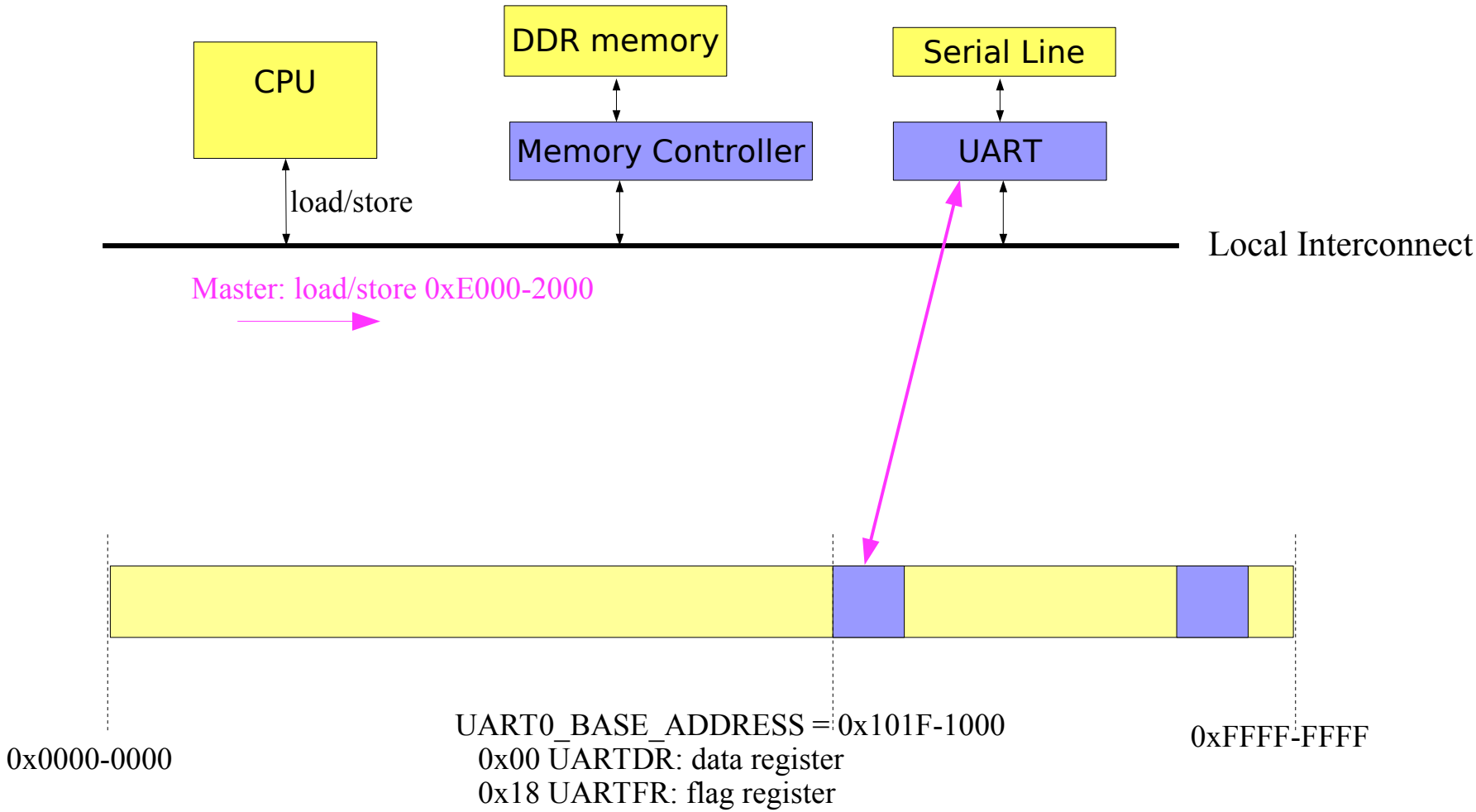
Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Bare Metal Programming – Outline

- Quick recap
  - Hardware concepts
  - Basic examples
- Identify main coding challenges
  - Power consumption, timing constraints, race conditions
  - Better engineering of the code
- Discussing interrupts
  - Hardware perspective
  - Software perspective

Hardware MMIO Registers

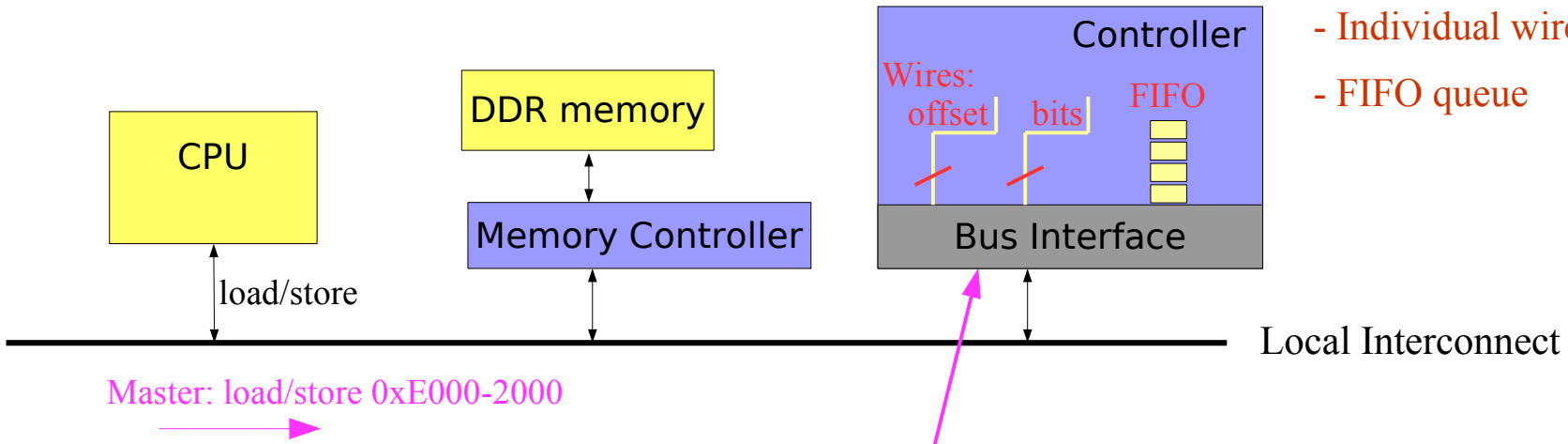


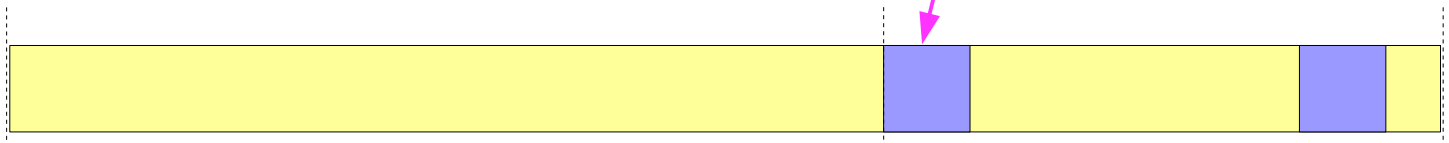
Memory-Mapped Input/Output Ranges

Hardware MMIO Registers

MMIO Registers:

- Individual wires
- FIFO queue





0x0000-0000

UART0_BASE_ADDRESS = 0x101F-1000
0x00 UARTDR: data register
0x18 UARTFR: flag register

0xFFFF-FFFF

Bare Metal Programming – A simple example

- A button/led example
  - A forever loop
  - Check button status → set LED on or off accordingly

```
void c_entry() {

  button_init_regs(BUTTON_MMIO_BASE_ADDRESS);
  led_init_regs(LED_MMIO_BASE_ADDRESS);

  for (;;) {
     uint8_t status = button_get_status(BUTTON_MMIO_BASE_ADDRESS);
     if (status & 0x01)
       led_on(LED_MMIO_BASE_ADDRESS);
      else
       led_off(LED_MMIO_BASE_ADDRESS);
  }
}
```

Challenge?

Bare Metal Programming – A simple example

- A button/led example
  - A forever loop
  - Check button status → set LED on or off accordingly

```
void c_entry() {

  button_init_regs(BUTTON_MMIO_BASE_ADDRESS);
  led_init_regs(LED_MMIO_BASE_ADDRESS);

  for (;;) {
     uint8_t status = button_get_status(BUTTON_MMIO_BASE_ADDRESS);
     if (status & 0x01)
       led_on(LED_MMIO_BASE_ADDRESS);
      else
       led_off(LED_MMIO_BASE_ADDRESS);
  }
}
```

Challenge: continuous polling.

→ processor never halts
→ high power consumption
→ short battery life

Bare Metal Programming – Another example

- A shell example via a serial line
  - Continuous polling of keyboard → high power consumption → short battery life
  - Other challenge such as timing constraints?

```
void c_entry() {

  uart_init_regs(UART0);

  unicode_t line[80];
  int offset = 0;
  for (;;) {
     uint8_t code = uart_get_code(UART0);
     if (code) {
       // send the corresponding character to the screen
       uart_put_byte(UART0, code);
       if ( code == '\n') { // do we have a full line?
         interpret(line,offset);
         offset = 0;
       } else
         line[offset++] = code;
     }
  }
}
```

Bare Metal Programming – Another example

- A shell example via a serial line
  - Continuous polling of keyboard → high power consumption → short battery life
  - Challenge: not loosing characters because of long commands

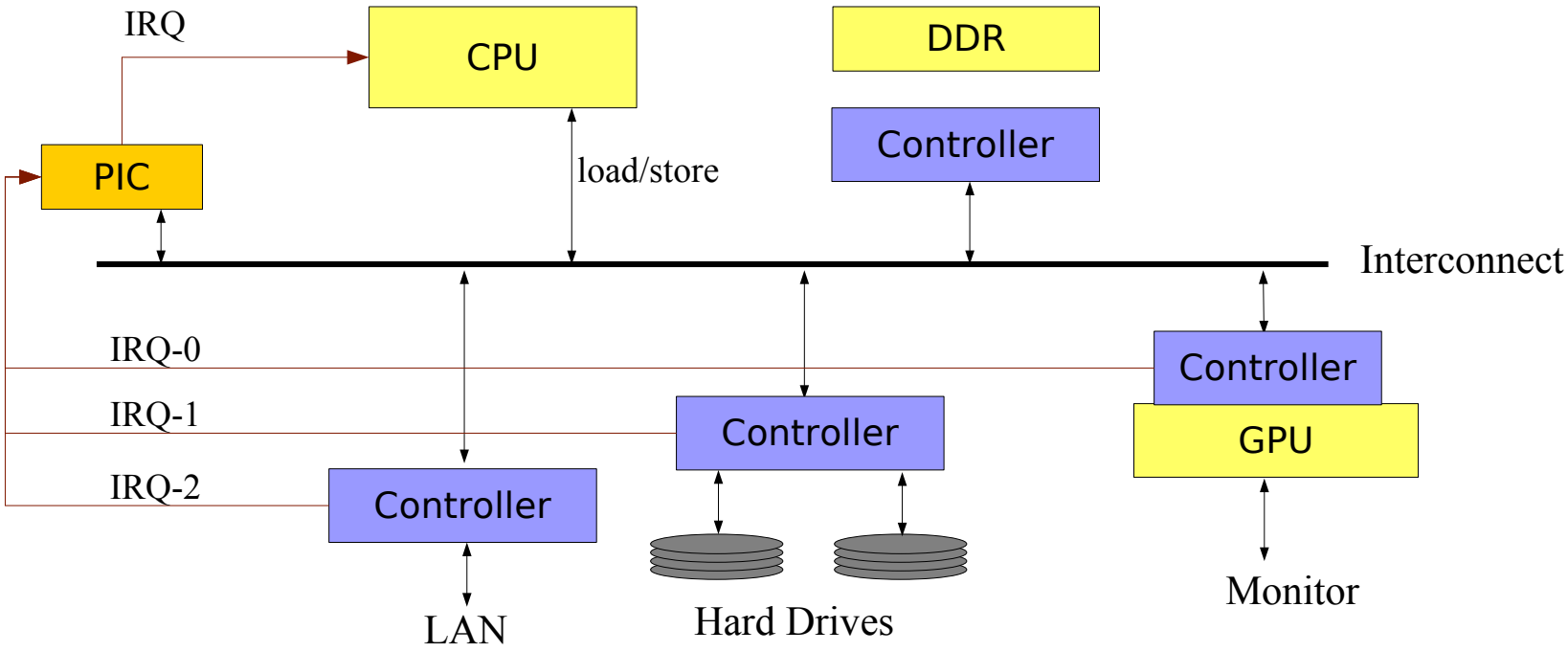
Evaluation:
UART with 8-byte FIFO
115200 bps → over 14000 Bps
One byte every 70 us
Dropping bytes after 560 us
560 us = (8 * 70us)

```
void c_entry() {

  uart_init_regs(UART0);

  unicode_t line[80];
  int offset = 0;
  for (;;) {
     uint8_t code = uart_get_code(UART0);
     if (code) {
       // send the corresponding character to the screen
       uart_put_byte(UART0, code);
       if ( code == '\n') { // do we have a full line?
         interpret(line,offset);
         offset = 0;
       } else
         line[offset++] = code;
     }
  }
}
```

Summary – Challenges

- Identified challenges
  - Continuous **polling**: waiting until a device has data available to read
  - Potential **spinning**: waiting until a device is ready to accept data
  - Timing constraints: limited time to pay attention to a device that requires attention
- Identified problems with our coding approach
  - Processor never halts, high power consumption, heat production
  - Potential loss of data if devices are not attended to in time
- Challenges
  - How do we allow the processor to halt?
  - **How do we know when a device needs attention?**

Interrupts – Hardware Perspective



PIC: Programmable Interrupt Controller
A chipset to memorize and multiplex interrupt requests from hardware devices
Configured and controlled through mmio registers, like any other controller

Interrupts – Hardware Perspective

- Device Controller Interrupts
  - Each device controller may raise one or more interrupts
  - Each interrupt is one dedicated electrical line to the PIC
- Programmable Interrupt Controller(1)
  - A new device on the bus, configured through mmio registers
  - Multiplexer/demultiplexer for IRQs from devices
- New processor pin
  - **Interrupt signal** from the PIC
  - Like a door bell, asking the processor to pay attention to one or more devices
  - The door bell can be enabled or disabled by software

(1) May have different names such as GIC or VIC or APIC

ARM Example – Hardware/Software Boundary

- Interrupts/Traps are ***exceptions*** for the processor

The 7 known exceptions:

```
.globl _vector = 0x00000000
_vector:
        ldr     pc, _reset
        ldr     pc, _undefined_instruction
        ldr     pc, _software_interrupt
        ldr     pc, _prefetch_abort
        ldr     pc, _data_abort
        ldr     pc, _not_used
        ldr     pc, _irq
        ldr     pc, _fiq

_reset:                     .word undefined_instruction
_undefined_instruction:     .word undefined_instruction
_software_interrupt:        .word software_interrupt
_prefetch_abort:            .word prefetch_abort
_data_abort:                .word data_abort
_not_used:                  .word not_used
_irq:                       .word _irq_handler
_fiq:                       .word fast_interrupt_service_routine

_irq_handler:
        …
        bl irq_handler
        ...
```

↔ ldr pc, [pc+18]

The irq_handler is the next processing step for handling interrupts...

ARM Example – Software perspective

Polling the PIC (via mmio reads)

Re-enabling interrupts at the PIC

```
void irq_handler(void) {
  for (;;) {
    int irq = pic_next_raised_irq();
    switch (irq) {
    case -1:
      pic_enable_all_irqs();
      return;
    case UART0_IRQ:
      uart0_isr();
      break;
    }
  }
}
```

The **interrupt service routine** for the UART0

Suppose to figure out why the UART0 raised its interrupt….

Here we assume there are pending characters

```
void uart0_isr() {
  ...
  uart_irq_ack(UART0); // device-specific
}
```

For most device, **must not** forget to acknowledge the IRQ to the their device

Questions

- PIC questions
  - How do we tell which interrupts has been raised?
  - How do we clear the interrupt status and why?
- UART questions
  - How do we tell which interrupts has been raised?
  - How do we clear the interrupt status and why?
- Hardware-software boundary questions
  - Processor mode when handling the interrupt
  - Assembly/C calling convention

ARM Example – Processor Modes

- ARM processors have different operating modes
  - **User**: normal program execution
  - **FIQ**: fast interrupt mode
  - **IRQ**: regular interrupt mode
  - **Supervisor**: protected mode for the operating system
  - **Abort**: when a page fault occurred when the MMU is turned on
  - **Undefined**: when an undefined instruction has encountered (supports emulation)
  - **System**: to run privileged operating system tasks
- User vs Privilege
  - The modes other than the User mode are considered privileged modes
  - They have full access to the processor and can reconfigure it

ARM Example – Processor Modes

	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

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

ARM Example – Interrupt Mode

When in interrupt mode → interrupts are disabled on the processor

Note that the interrupt mode has its own **stack**, **link**, and **SPSR** registers.

R0 – R12: general-purpose registers
R13: sp (stack pointer)
R14: lr (link register)
R15: pc (program counter)

CPSR: Current Program Status Register
SPSR: Saved Program Status Register

The Current Program Status Register (CPSR) is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information.

Each exception mode also has a Saved Program Status Register (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs.

Interrupt	
R0	
R1	
R2	
R3	
R4	
R5	
R6	
R7	
R8	
R9	
R10	
R11	
R12	
R13_irq	(banked)
R14_irq	(banked)
PC	

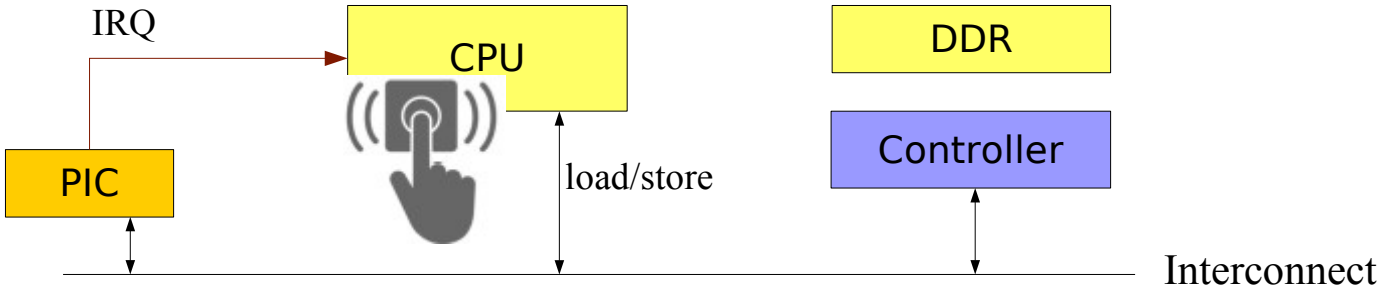
CPSR	
SPSR_irq	(banked)

ARM Example – IRQ Setup

Startup initialization:

- Setting up the IRQ-mode stack
  (define in the linker script as another stack)
- Enabling the IRQ interrupts at the processor[1]

```
.global _setup_irqs
_enable_irq:
    /* get Program Status Register */
    MRS r0, cpsr
    /* go in IRQ mode */
    BIC r1, r0, #0x1F
    ORR r1, r1, #0x12
    MSR cpsr, r1
    /* set IRQ stack */
    LDR sp, =irq_stack_top
    /* Enable IRQs */
    BIC r0, r0, #0x80
    /* go back in Supervisor mode */
    MSR cpsr, r0
    mov pc,lr
```



(1) Meaning the PIC can ring the door on processor and asking the processor to handle an interrupt. The processor will do it, eventually, but not immediately.

ARM Example – IRQ Handler


```
_irq_handler:
     sub lr,lr,#4
     stmfd sp!, {r0-r12,lr}
     bl irq_handler
     ldmfd sp!, {r0-r12,pc}^


void irq_handler() {

}
```

IRQ handler:

- Fix the link register
- Save all non-banked registers and link register
- Upcall the C function
- Restore all non-banked registers and the pc

Note:

The ^ symbol at the end of the LDMFD instruction
means that the CPSR will be restored from the save SPSR.
This only happens if the pc is loaded by the LDMFD instruction.

Restoring the SPSR means re-enabling the interrupts on the processor.
Indeed, if we are processing an interrupt, they were enabled on the processor.
By going in interrupt mode, the processor disabled further interrupts.
By returning from interrupt mode, the processor therefore restores the ability to handle further interrupts.

(1) Meaning the PIC can ring the door on processor and asking it to handle an interrupt

Questions

- PIC questions
  - How do we tell which interrupts has been raised?
  - How do we clear the interrupt status and why?
- UART questions
  - How do we tell which interrupts has been raised?
  - How do we clear the interrupt status and why?

- ISA-related questions
  - Processor mode when handling the interrupt
  - Assembly/C calling convention

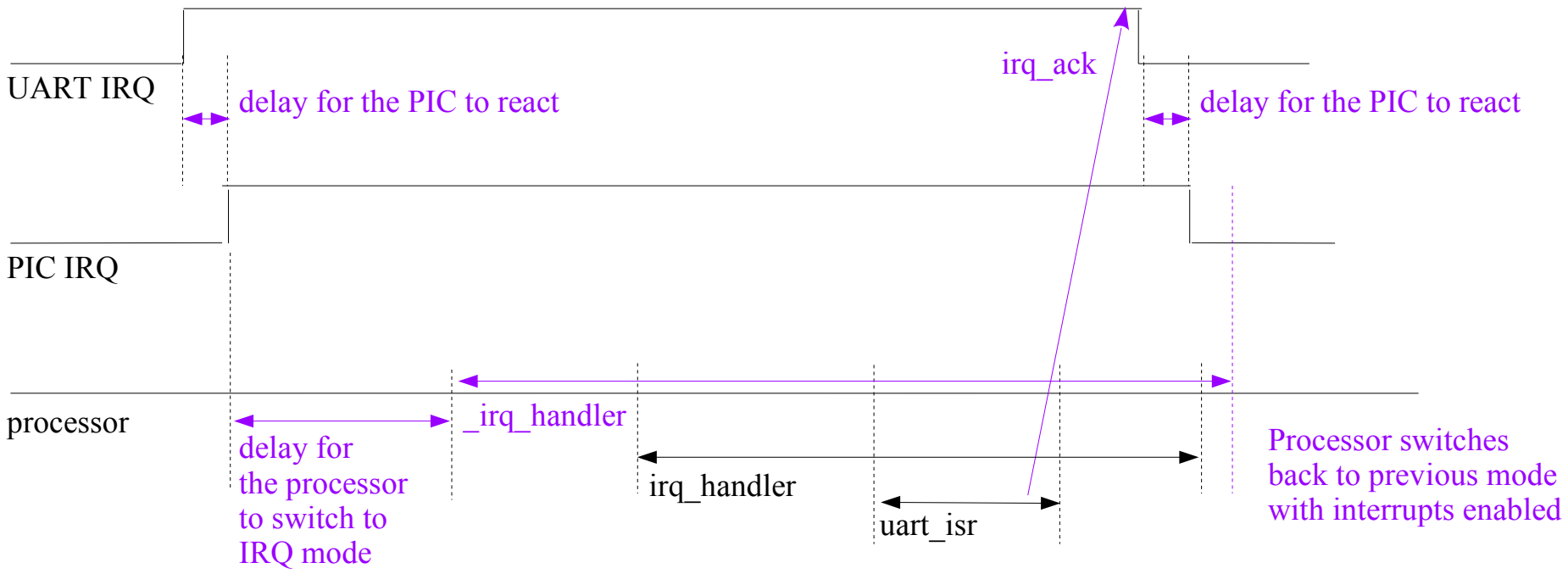
UART Management – PL011[1] Example

- How do we tell which interrupts has been raised?
  - The UARTIMSC Register is the interrupt mask **set/clear**[2] register. It is a read/write register. ***Masked interrupts are enabled***
  - The UARTRIS Register is the raw interrupt status register. It is a read-only register that returns the current raw status value, prior to masking
  - The UARTMIS Register is the masked interrupt status register. It is a read-only register. This register returns the current masked status value of the corresponding interrupt. A write has no effect
- How do we clear the interrupt status at the UART?
  - The UARTICR Register is the interrupt clear register and is write-only. On a write of 1, the corresponding interrupt is cleared. A write of 0 has no effect.
  - **Note:** the **receive interrupt** is also cleared by performing at least one read of the receive FIFO, it may be cleared by clearing the interrupt in UARTICR register.

(1) PrimeCell UART (PL011) Technical Reference Manual (DDIO183G)
(2) Set/Clear register: write 1s to set the corresponding mask of that interrupt, write 0s to clear it

UART Management – PL011 Example

- Why do we need to clear the interrupt at the UART (PL011)?
  - How would the UART otherwise know when the interrupt has been handled by software?
  - It needs to know in order to know when it can raise another interrupt if necessary



Questions

- PIC questions
  - How do we tell which interrupts has been raised?
  - How do we clear the interrupt status and why?

- UART questions
  - How do we tell which interrupts has been raised?
  - How do we clear the interrupt status and why?
- ISA-related questions
  - Processor mode when handling the interrupt
  - Assembly/C calling convention

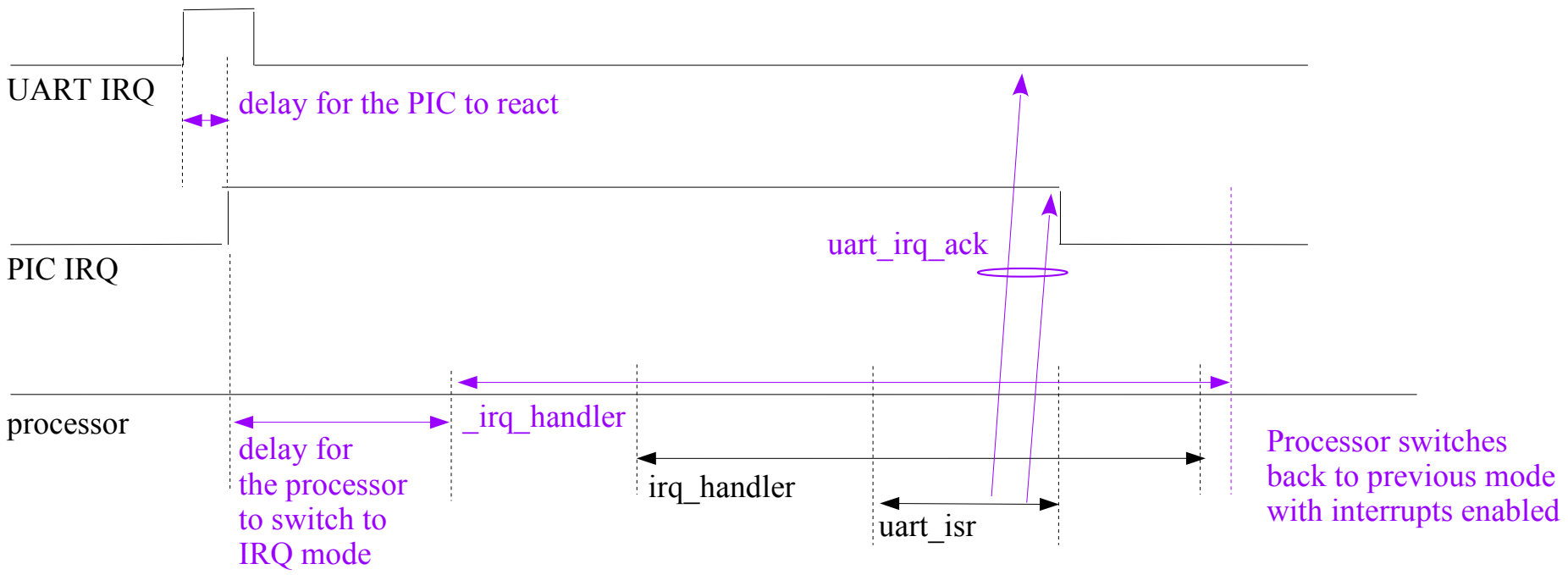
PIC Management – PL190[1] Example

- How do we tell which interrupts has been raised?
  - The read-only VIC_IRQ_STATUS[2] register provides the status of interrupts [31:0] after IRQ masking. **Masked interrupts are enabled**.
  - The read/write VIC_INT_ENABLE[2] register enables the interrupt request lines, by masking the interrupt sources for the IRQ interrupt.
  - The write-only VIC_INT_ENCLEAR[2] register clears bits in the VIC_INT_ENABLE register.
- How do we clear the interrupt status at the VIC?
  - Not necessary for the PL190 because interrupt inputs must be level sensitive, active HIGH, and held asserted until the interrupt service routine clears the interrupt.
  - For PIC that handles edge-triggered interrupts, an ACK is usually necessary for the same reason as for the UART, the PIC needs to know when a particular interrupt has been handled by software

(1) PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual
(2) Named in the doc VICIRQSTATUS, VICINTENABLE, VICINTENCLEAR

PIC Management – Edge-Triggered

Example of a PIC that handles edge-triggered IRQs from controllers
and allowing individual acknowledgment of such IRQs,
(typically the case on Intel boards and the Advanced Programmable Interrupt Controller (APIC))



Summary – Challenges

- Identified challenges
  - Continuous **polling**: waiting until a device has data available to read
  - Potential **spinning**: waiting until a device is ready to accept data
  - Timing constraints: limited time to pay attention to a device that requires attention
- Identified problems with our coding approach
  - Processor never halts, high power consumption, heat production
  - Potential loss of data if devices are not attended to in time
- Challenges
  - **How do we allow the processor to halt?**
  - How do we know when a device needs attention?

Interrupts – Software perspective

- New instruction: **halt**[1]
  - Enables interrupts
  - Halts only if the processor interrupt signal is inactive
  - Wakes up when the processor interrupt signal becomes active

```
void uart0_isr() {
  ...
  uart_irq_ack(UART0);
}

void main() {
  uart_init_regs(UART0);
  for (;;) {
    ...
    wfi();
  }
}
```

Don‘t forget to halt the processor when there is nothing more to do… Here, we ***W***ait ***F***or ***I***nterrupts (wfi)

(1): Halt instruction has different names for different processors
May not be an instruction on some processor, but a flag in a control register:

```
ARMv5:    MCR p15,0,r0,c7,c0,4
ARMv7:    wfi
Intel:    htl
```

Interrupts – Software perspective

- New instruction: **halt**
  - Enables interrupts
  - Halts only if the processor interrupt signal is inactive
  - Wakes up when the processor interrupt signal becomes active

```
void uart0_isr() {
  ...
  uart_irq_ack(UART0);
}

void main() {
  uart_init_regs(UART0);
  for (;;) {
    ...
    wfi();
  }
}
```

Missing something, are we not?

We need a communication channel between the ISR and the main loop…

Right?
- The ISR gathers the characters…
- The main processes the characters...

Our Shell Example with IRQs


```
uint8_t line[80];
int offset = 0;
int full_line = 0;

void uart0_isr() {
  uint8_t code = uart_get_byte(UART0);
  while (code) {
    if ( code == '\n') { // full line?
      full_line = 1;
      break;
    } else
      line[offset++] = code;
    code = uart_get_byte(UART0);
  }
  uart_irq_ack(UART0);
}

void main() {
  uart_init_regs(UART0);
  for (;;) {
     if (full_line) {
       interpret(line,offset);
       offset = 0;
       full_line = 0;
     }
     halt();
  }
}
```

Looks good… right?

We introduced a buffer to gather the characters from the UART, one line at a time.

Our Shell Example with IRQs


```
uint8_t line[80];
int offset = 0;
int full_line = 0;

void uart0_isr() {
  uint8_t code = uart_get_byte(UART0);
  while (code) {
    if ( code == '\n') { // full line?
      full_line = 1;
      break;
    } else
      line[offset++] = code;
    code = uart_get_byte(UART0);
  }
  uart_irq_ack(UART0);
}

void main() {
  uart_init_regs(UART0);
  for (;;) {
     if (full_line) {
       interpret(line,offset);
       offset = 0;
       full_line = 0;
     }
     halt();
  }
}
```

Not really...

Even though it may work for a while…

It may run, but it is not correct!

Do you see what is the problem?

Our Shell Example with IRQs


```
uint8_t line[80];
int offset = 0;
int full_line = 0;

void uart0_isr() {
  uint8_t code = uart_get_byte(UART0);
  while (code) {
    if ( code == '\n') { // full line?
      full_line = 1;
      break;
    } else
      line[offset++] = code;
    code = uart_get_byte(UART0);
  }
  uart_irq_ack(UART0);
}

void main() {
  uart_init_regs(UART0);
  for (;;) {
     if (full_line) {
       interpret(line,offset);
       offset = 0;
       full_line = 0;
     }
     halt();
  }
}
```

Race condition between:
(a) UART ISR
(b) Line parsing to interpret

over the shared line buffer!

Our Shell Example with IRQs – First Solution


```
uint8_t line[80];
int offset = 0;
int full_line = 0;

void uart0_isr() {
  uint8_t code = uart_get_byte(UART0);
  while (code) {
    if ( code == '\n') { // full line?
      full_line = 1;
      pic_disable_irqs();
      break;
    } else
      line[offset++] = code;
    code = uart_get_byte(UART0);
  }
  uart_irq_ack(UART0);
}

void main() {
  uart_init_regs(UART0);
  for (;;) {
     if (full_line) {
       interpret(line,offset);
       offset = 0;
       full_line = 0;
       pic_enable_irqs();
     }
     halt();
  }
}
```

First solution → disable/enable IRQs

- Disable IRQs when a full line has been captured,
- Re-enable IRQs when the full line has been interpreted.

Challenge?

Our Shell Example with IRQs – Second Solution


```
#define MAX_CHARS 256
volatile uint8_t buffer[MAX_CHARS];
volatile int head = 0;
volatile int tail = 0;

void uart0_isr() {
  uint8_t code = uart_get_byte(UART0);
  while (code) {
    int next = (head + 1) % MAX_CHARS;
    if (next==tail) return;
    buffer[head] = code;
    head = next;
    code = uart_get_byte(UART0);
  }
}

void main() {
  int eol = 0;
  uart_init_regs(UART0);
  for (;;) {
     while (eol!=head) {
       eol = (eol + 1) % MAX_CHARS;
       if (buffer[eol] == '\n') {
         interpret(buffer,tail,eol);
         eol = (eol + 1) % MAX_CHARS;
         tail = eol;
       }
     }
     halt();
  }
}
```

Second solution → circular buffer

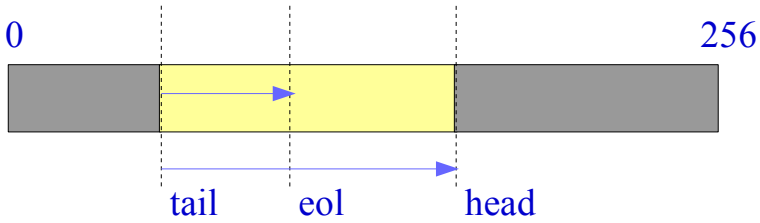
Note: a lock-free design is necessary

Why? Because blocking the ISR would in fact deadlock the execution!

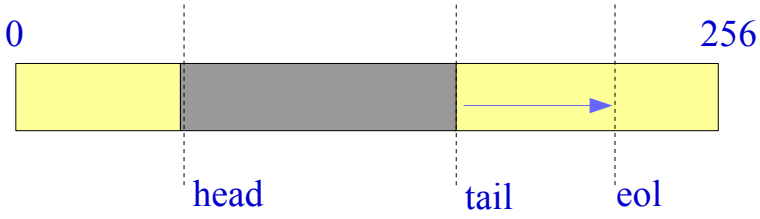
Do you see why?

Using a Lock-free Circular Buffer


```c
#define MAX_CHARS 256
volatile uint8_t buffer[MAX_CHARS];
volatile int head = 0;
volatile int tail = 0;

void uart0_isr() {
  uint8_t code = uart_get_byte(UART0);
  while (code) {
    int next = (head + 1) % MAX_CHARS;
    if (next==tail) return;
    buffer[head] = code;
    head = next;
    code = uart_get_byte(UART0);
  }
}

void main() {
  int eol = 0;
  uart_init_regs(UART0);
  for (;;) {
     while (eol!=head) {
       eol = (eol + 1) % MAX_CHARS;
       if (buffer[eol] == '\n') {
         interpret(buffer,tail,eol);
         eol = (eol + 1) % MAX_CHARS;
         tail = eol;
       }
     }
     halt();
  }
}
```



Circular Buffer:
Acts as a lock-free communication channel between the ISR and the main loop



Using a Lock-free Circular Buffer


```
#define MAX_CHARS 256
volatile uint8_t buffer[MAX_CHARS];
volatile int head = 0;
volatile int tail = 0;

void uart0_isr() {
  uint8_t code = uart_get_byte(UART0);
  while (code) {
    int next = (head + 1) % MAX_CHARS;
    if (next==tail) return;
    buffer[head] = code;
    head = next;
    code = uart_get_byte(UART0);
  }
}

void main() {
  int eol = 0;
  uart_init_regs(UART0);
  for (;;) {
     while (eol!=head) {
       eol = (eol + 1) % MAX_CHARS;
       if (buffer[eol] == '\n') {
         interpret(buffer,tail,eol);
         eol = (eol + 1) % MAX_CHARS;
         tail = eol;
       }
     }
     halt();
  }
}
```

Circular Buffer

Warning: the circular buffer may be full…

Can you loose characters then? Of course…

This means that the main loop must be able to keep up with incoming characters before the circular buffer fills up

There is no free lunch!

The circular buffer only buys your code more time to react...

Code Engineering – Quite Necessary

- We strongly suggest the following
  - Clearly separate the core exception and IRQ support
    - Exception vector
    - Global interrupt handler
  - Clearly organize the UART code (uart.c and uart.h)
    - Including the UART interrupt service routine
    - Circular buffer
- Cleaner code / easier development
  - Easier to read
  - Easier to understand
  - Easier to debug
  - Easier to evolve

ARM Documentation

- Architecture Reference Manual
  - For the processor, the VersatilePB uses a ARMv5 processor
- Technical Reference Manuals
  - PrimeCell Vectored Interrupt Controller (PL190)
  - PrimeCell UART (PL011)
- How to read them?
  - (1) do not get scared, do not run away to Google land…
    - Understand they target two audiences: hardware integrators and software developers
    - So focus on the parts that are for software developers
  - (2) understand the typical outline
    - Functional Overview ← more hardware than software
    - Programmers Model ← more software than hardware