M2PGi – M2M

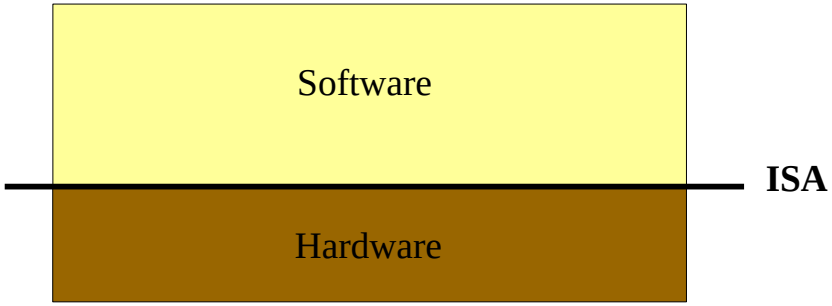
Pr. Olivier Gruber (olivier.gruber@imag.fr)

Laboratoire d'Informatique de Grenoble

Université de Grenoble-Alpes

Today

- Hardware Support for Interrupts
  - Halt instruction, Processor modes, Interrupt Requests, Exception vectors, etc.
- Software Support for Interrupts
  - Handling interrupts
  - Race condition challenge



Analysis – Button & LED Example


```
uint8_t button_get_status(void* bar);
void led_on(void* bar);
void led_off(void* bar);


void start() {

  button_init_regs(BUTTON_BAR);
  led_init_regs(LED_BAR);
  for (;;) {
     uint8_t status = button_get_status(BUTTON_BAR);
     if (status & 0x01)
       led_on(LED_BAR);
     else
       led_off(LED_BAR);
  }
}
```

Does it work? Sure.
But how well?

Analysis – Button & LED Example


```
uint8_t button_get_status(void* bar);
void led_on(void* bar);
void led_off(void* bar);


void start() {

  button_init_regs(BUTTON_BAR);
  led_init_regs(LED_BAR);
  for (;;) {
     uint8_t status = button_get_status(BUTTON_BAR);
     if (status & 0x01)
       led_on(LED_BAR);
     else
       led_off(LED_BAR);
  }
}
```

Problem:

continuous polling/spinning.

→ processor never halts
→ high power consumption

Limitation:

our software does only one thing/task, what if we had multiple tasks to carry out?

Analysis – The Console

How well does this code work?

Think timing...

```
void _start() {

  uart_init(UART0);

  for (;;) {
    // read available byte from the keyboard
    uint8_t code = uart_receive(UART0);

    // echo the character to the terminal
    uart_send(UART0, code);

  }
}
```

Analysis – The Console

How well does this code work?

Think timing... well...
It is *spinning most of the time*,
waiting for bytes to be received
by the UART...

```
void _start() {

  uart_init(UART0);

  for (;;) {
    // read available byte from the keyboard
    uint8_t code = uart_receive(UART0);

    // echo the character to the terminal
    uart_send(UART0, code);

  }
}
```

Analysis – Console & Shell


```
void shell(char* line, int length);

char line[80];
int offset = 0;

void _start() {

  uart_init(UART0);
  for (;;) {
    uint8_t code = uart_receive(UART0);
    while (code) {
      uart_send(UART0, code);
      if (code == '\n') {
        shell(line, offset);
        offset=0;
      } else
        line[offset++]=(char)code;
      code = uart_receive(UART0);
    }
  }
}
```

Adding a shell that interprets command lines entered at the console...

Problems?

Analysis – Console & Shell

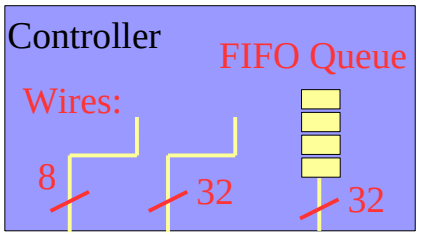

```
void shell(char* line, int length);

char line[80];
int offset = 0;

void _start() {

  uart_init(UART0);
  for (;;) {
    uint8_t code = uart_receive(UART0);
    while (code) {
      uart_send(UART0, code);
      if (code == '\n') {
        shell(line, offset);
        offset=0;
      } else
        line[offset++]=(char)code;
      code = uart_receive(UART0);
    }
  }
}
```

UART with 8-byte FIFO
115200 bps → over 14000 Bps
One byte every 70 us

*Dropping bytes after **560 us***
560 us = (8 * 70us)

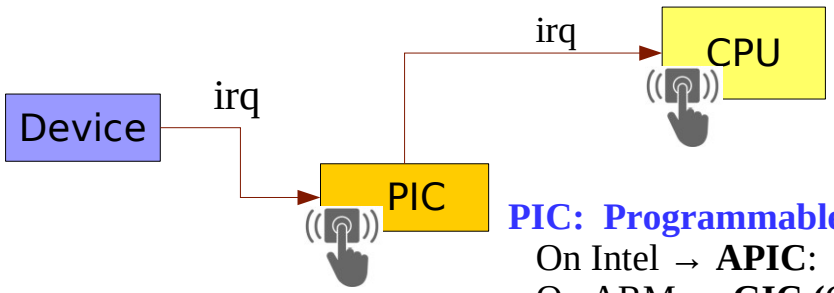
Embedded ARM Core @ 30MHz
1 cycle = 1 instruction
560 us = 560*30 = 16800 instructions

Nota Bene: timing matters...
this code may loose received bytes!



Hardware – Interrupt Support

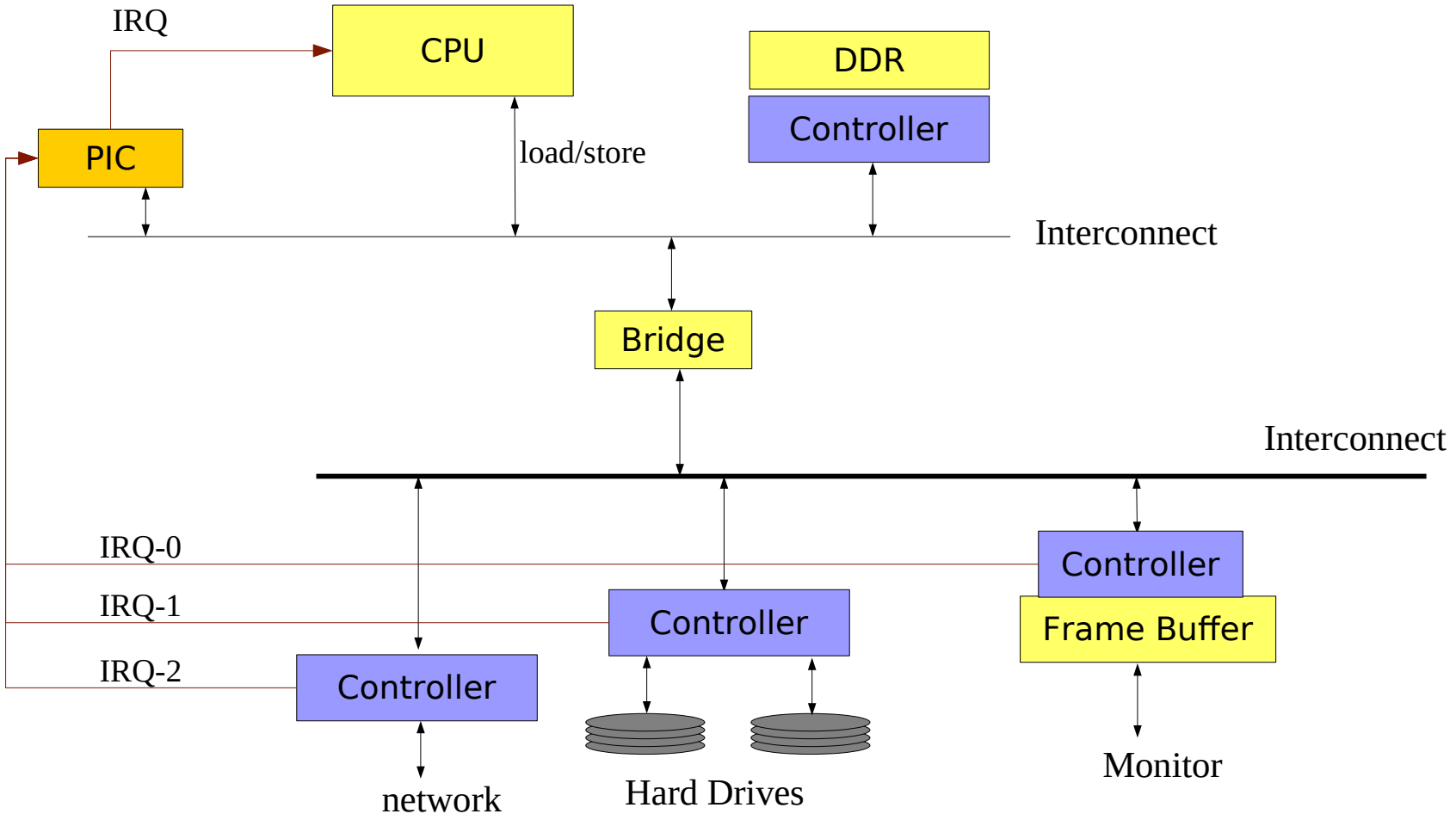
- Processor support
  - New instruction: halt[(1)(2)]
  - New processor pin: interrupt signal, enabled or disabled
- Programmable Interrupt Controller (PIC)
  - A new device on the bus, configured through mmio registers
  - Multiplexer/demultiplexer for IRQs from devices



PIC: Programmable Interrupt Controller
On Intel → **APIC**: Advanced Programmable Interrupt Controller
On ARM → **GIC (**Generic Interrupt Controller)
or **VIC** (Vectored Interrupt Controller)

(1): Halt instruction has different names for different processors
For example: WFI or WFE for ARM processors
(2): Halts the processor only if there are no pending interrupts
and wakes up as soon as an interrupt is pending.

Hardware – Interrupt Support

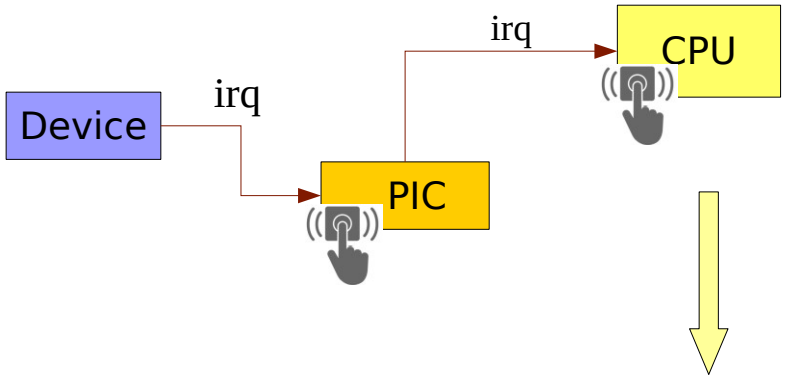


PIC: Programmable Interrupt Controller

A chipset to memorize and multiplex interrupt requests from hardware devices.

Configured and controlled through mmio registers, like any other controller

Idea – Using interrupts




```
void button_interrupt_handler() {
  uint8_t status = button_get_status(BUTTON_MMIO_BAR);
  if (status & 0x01)
    led_on(LED_MMIO_BAR);
  else
    led_off(LED_MMIO_BAR);
}

void start() {
  button_init_regs(BUTTON_MMIO_BAR);
  led_init_regs(LED_MMIO_BAR);

  for (;;) {
    halt();
  }
}
```

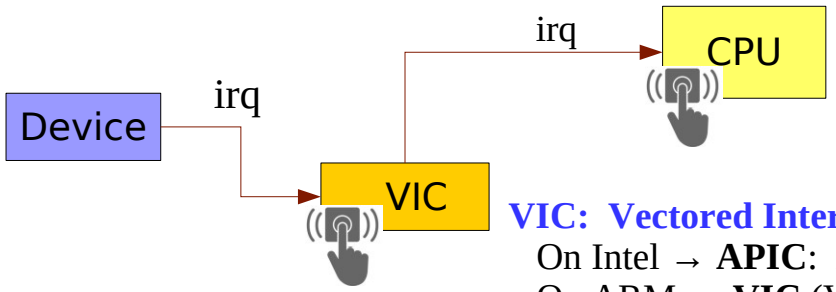
Function to be called when the button is pressed, and only then.

But how will that work exactly?

Discussing Interrupts

- When should the processor react to interrupts?
- What should happen?

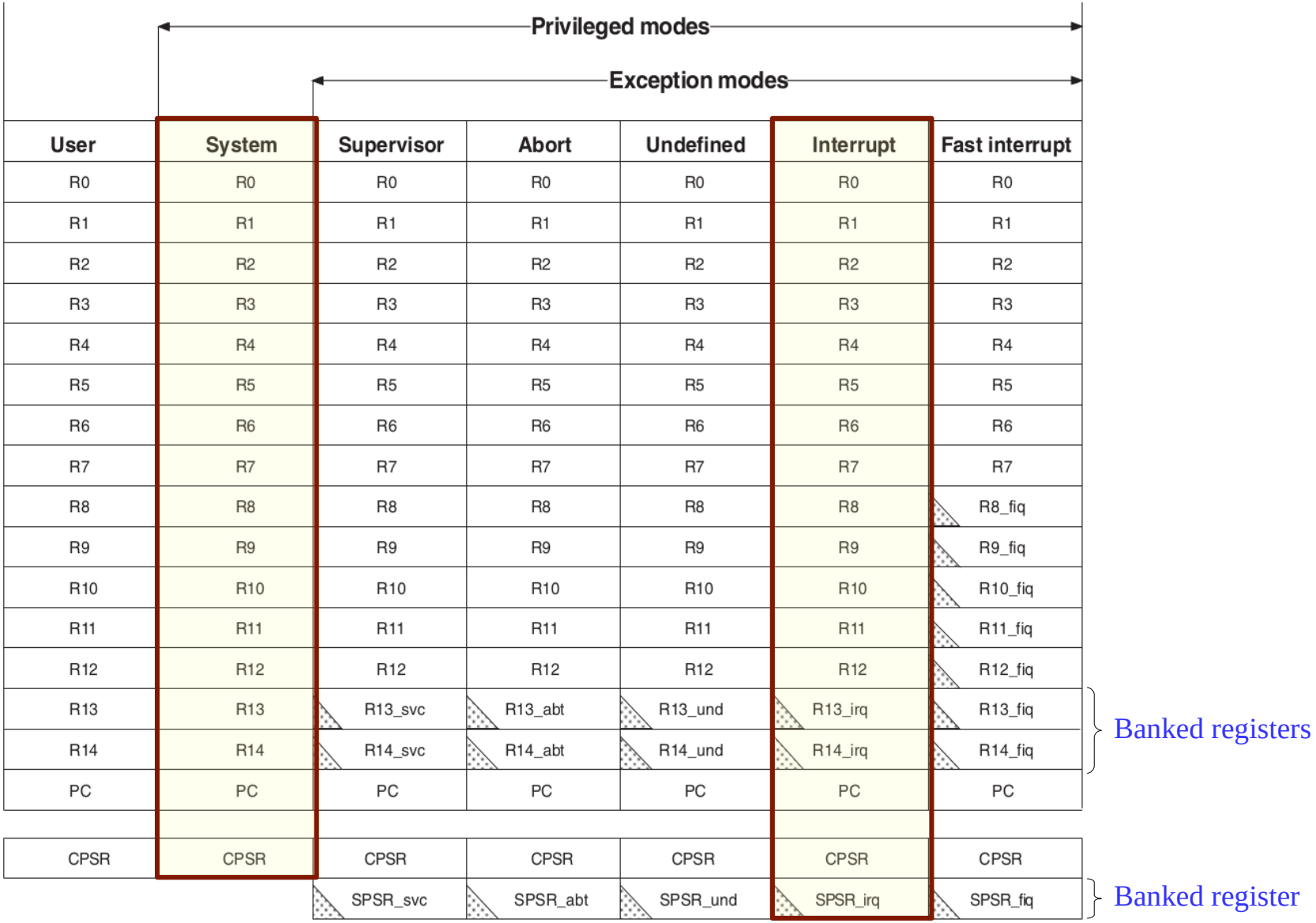
It will be a cooperation between software and hardware



VIC: Vectored Interrupt Controller

On Intel → **APIC**: Advanced Programmable Interrupt Controller
On ARM → **VIC** (Vectored Interrupt Controller)
or **GIC (**Generic Interrupt Controller)

Processor Modes – ARM Example



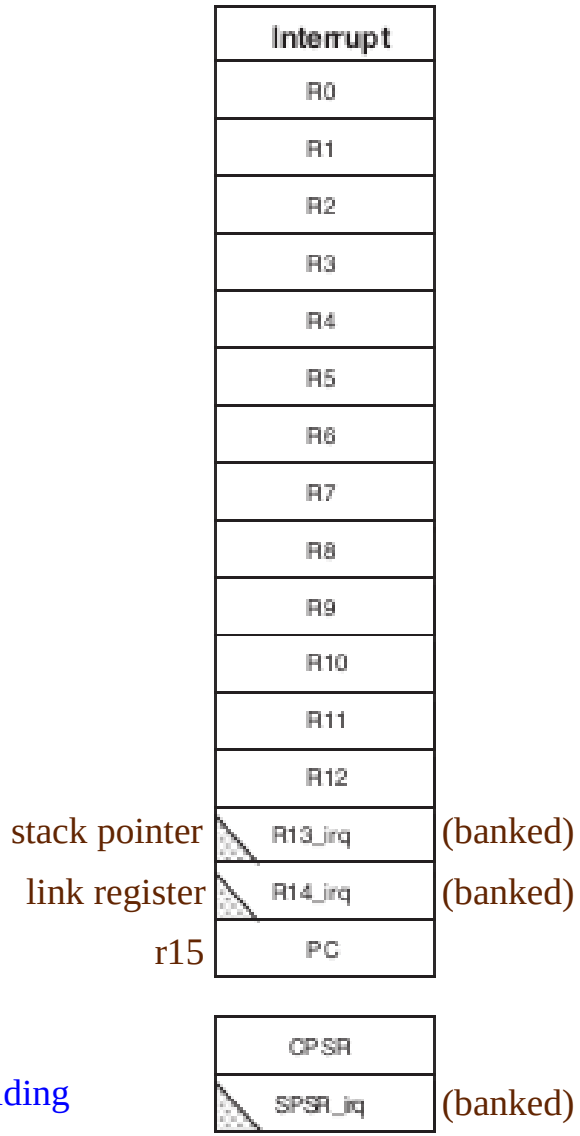
Interrupt Processing – Processor perspective

Interrupt steps taken by the processor:

- Save the CPSR into the banked ***SPSR_irq***
- Save **R15** into the banked ***R14_irq***
- Switch the processor in ***interrupt mode, with disabled interrupts***
- Set **R15** to **0x18** and execute the instruction there

But what is the value of the banked register R13_irq (sp)?

- It must be set as part of the system initialization
- Done after a hardware reset
- There is a system register that controls the current mode.



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

One of the most important system registers, controlling your processor and providing status bits. In particular, the current operation mode of your processor.

Interrupt Processing – Hardware / Software Boundary

- Exception vector in memory
  - Interrupts are “one kind of exceptions”
  - Failures are others

This is reset

These are traps, as a side effect of executing an instruction.

This is for hardware interrupts from devices

```
; hardware reset address = 0x00000000

0x00    br   _reset
0x04    br   _undef_inst
0x08    br   _soft_irq
0x0c    br   _prefetch_abort
0x10    br   _data_abort
0x14    br   _not_used
0x18    br   _isr
0x1c    br   _fisr
```

traps

This is the software reaction/continuation to the hardware having reset...

```
_reset:
   ...
```

The *Interrupt Service Routine* processes all interrupts...

```
_isr:
   ...
```

Interrupt Processing – Hardware / Software Boundary

More flexible coding of the same thing:

- Use a special load instruction, relative to the address in the pc register

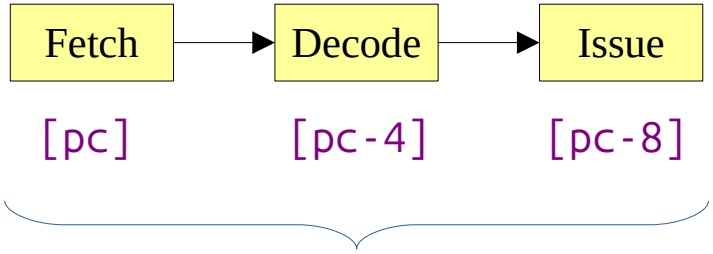
  `ldr pc, [ pc, #offset ]`

- Removes the constraint of having the called function near by, because a branch instruction can have only a small offset encoded in the instruction.

Nota Bene:

Because of the ***3-stage pipeline***, the pc is 8 bytes ahead on Cortex-A8,
hence: `ldr pc, [pc,#0x18]`
and not: `ldr pc, [pc,#0x20]`

```
; hardware reset address = 0x00000000

0x00    ldr  pc, [pc,#0x18]
0x04    ldr  pc, [pc,#0x18]
0x08    ldr  pc, [pc,#0x18]
0x0c    ldr  pc, [pc,#0x18]
0x10    ldr  pc, [pc,#0x18]
0x14    ldr  pc, [pc,#0x18]
0x18    ldr  pc, [pc,#0x18]
0x1c    ldr  pc, [pc,#0x18]

0x20    .word _reset
0x24    .word _undefined_instruction
0x28    .word _software_interrupt
0x2c    .word _prefetch_abort
0x30    .word _data_abort
0x34    .word not_used
0x38    .word _isr
0x3c    .word _fisr
```




```
Issue @[pc-8]   0x10000   mov  r0, #0x24
Decode @[pc-4]  0x10004   ldrb  r1, [r0]
Fetch @[pc]     0x10008   mov  r2, #0x28
                0x1000C   str   r1, [r2]
```

Interrupt Service Routine

- Interrupt Service Routine (ISR)
  - The simplest interrupt service routine, up-calling C code
  - Then returning to normal execution

```
_isr: ; interrupt service routine
    sub lr,lr,#4                    ; adjust return address (see Cortex-A8 docs)
    stmfd   sp!, {r0-r12, lr}       ; save registers with link register last
    bl isr                          ; now call the C function interrupt_service_routine
    ldmfd    sp!, {r0-r12, pc}^     ; back from C…
                                    ; restore all registers, including pc from saved lr
```

Note: use store/load multiple instructions (stm/ldm) for stack manipulations,
stacks can be descending (stmfd/ldmfd) or ascending (stmfa/ldmfa), stack pointer (sp) is r13.
The presence of '!' specifies that the final address is written back into r13.

Note: ***The ^ qualifier on ldmfd specifies that the CPSR is restored from the SPSR***.
CPSR: Current Program Status Register.
SPSR: Saved Program Status Register.

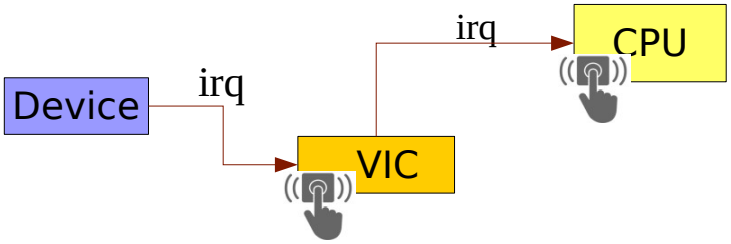
Interrupt Service Routine

- Cortex-a8 (ARM DDI 0344E), page 2-35

Table 2-12 Exception entry and exit

Exception or entry	Return instruction	Previous state		Notes
		ARM r14_x	Thumb r14_x	
SVC	MOVS PC, R14_svc	PC + 4	PC+2	Where the PC is the address of the SVC, SMC, or Undefined instruction
SMC	MOVS PC, R14_mon	PC + 4	-	
UNDEF	MOVS PC, R14_und	PC + 4	PC+2	
PABT	SUBS PC, R14_abt, #4	PC + 4	PC+4	Where the PC is the address of instruction that had the prefetch abort
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC+4	Where the PC is the address of the instruction that was not executed because the FIQ or IRQ took priority
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC+4	
DABT	SUBS PC, R14_abt, #8	PC + 8	PC+8	Where the PC is the address of the load or store instruction that generated the data abort
RESET	-	-	-	The value saved in r14_svc on reset is Unpredictable
BKPT	SUBS PC, R14_abt, #4	PC + 4	PC+4	Software breakpoint

Summary



The instruction "halt" such as ***WFI*** or ***WFE*** for ARM processors

Vectored Interrupt Controller: (architected state per interrupt)

Trapping for the Interrupt Processing by the processor

FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC+4	Where the PC is the address of the instruction that was not executed because the FIQ or IRQ took priority
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC+4	

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

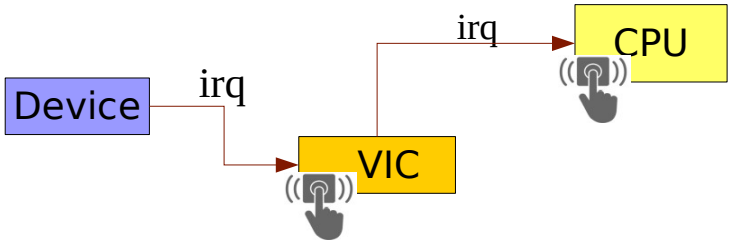
```
0x00    br   _reset
0x04    br   _undef_inst
0x08    br   _soft_irq
0x0c    br   _prefetch_abort
0x10    br   _data_abort
0x14    br   _not_used
0x18    br   _isr
0x1c    br   _fisr
```

Enabling Interrupts

Enabling at the device…

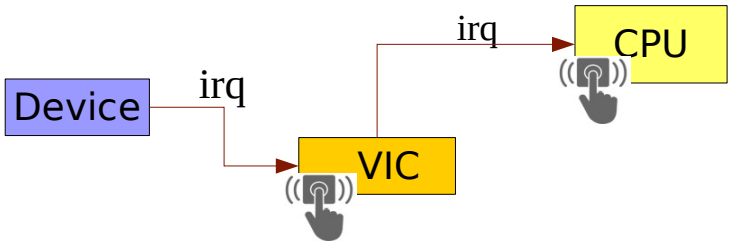
Enabling at the VIC…

Enabling at the CPU (rather at the core level)



Enabling Interrupts

Enabling at the device…



PrimeCell UART (PL011) Technical Reference Manual

Chapter 3 **Programmers Model**

Contents

PL190 Registers – Data & Status

3.2 Summary of registers

Table 3-1 lists the UART registers.

Table 3-1 UART register summary

Offset	Name	Type	Reset	Width	Description
0x000	UARTDR	RW	0x---	12/8	*Data Register, UARTDR* on page 3-5
0x004	UARTRSR/ UARTECR	RW	0x0	4/0	*Receive Status Register/Error Clear Register, UARTRSR/UARTECR* on page 3-6
0x008-0x014	-	-	-	-	Reserved
0x018	UARTFR	RO	0b-10010---	9	*Flag Register, UARTFR* on page 3-8
0x01C	-	-	-	-	Reserved

PL190 Registers – Interrupt-related Registers

0x038	UARTIMSC	RW	0x000	11	*Interrupt Mask Set/Clear Register, UARTIMSC* on page 3-17
0x03C	UARTRIS	RO	0x00-	11	*Raw Interrupt Status Register, UARTRIS* on page 3-19
0x040	UARTMIS	RO	0x00-	11	*Masked Interrupt Status Register, UARTMIS* on page 3-20
0x044	UARTICR	WO	-	11	*Interrupt Clear Register, UARTICR* on page 3-21

PrimeCell UART (PL011) Technical Reference Manual

Interrupt Mask Set/Clear Register (UARTIMSC)

The UARTIMSC Register is the interrupt mask ***set/clear*** register.

It is read/write register:

- On a read this register returns the current value of the mask on the relevant interrupt.
- On a write of 1 to a particular bit, it sets the corresponding mask of that interrupt.
- On a write of 0 to a particular bit, it clears the corresponding mask.

All the bits are cleared to when reset, so all interrupts are disabled after a reset.

PL190 Registers – UARTIMSC Bit Field

Table 3-14 UARTIMSC Register

Bits	Name	Function
15:11	-	Reserved, read as zero, do not modify.
10	OEIM	Overrun error interrupt mask. A read returns the current mask for the **UARTOEINTR** interrupt. On a write of 1, the mask of the **UARTOEINTR** interrupt is set. A write of 0 clears the mask.
9	BEIM	Break error interrupt mask. A read returns the current mask for the **UARTBEINTR** interrupt. On a write of 1, the mask of the **UARTBEINTR** interrupt is set. A write of 0 clears the mask.
8	PEIM	Parity error interrupt mask. A read returns the current mask for the **UARTPEINTR** interrupt. On a write of 1, the mask of the **UARTPEINTR** interrupt is set. A write of 0 clears the mask.
7	FEIM	Framing error interrupt mask. A read returns the current mask for the **UARTFEINTR** interrupt. On a write of 1, the mask of the **UARTFEINTR** interrupt is set. A write of 0 clears the mask.
6	RTIM	Receive timeout interrupt mask. A read returns the current mask for the **UARTRTINTR** interrupt. On a write of 1, the mask of the **UARTRTINTR** interrupt is set. A write of 0 clears the mask.
5	TXIM	Transmit interrupt mask. A read returns the current mask for the **UARTTXINTR** interrupt. On a write of 1, the mask of the **UARTTXINTR** interrupt is set. A write of 0 clears the mask.
4	RXIM	Receive interrupt mask. A read returns the current mask for the **UARTRXINTR** interrupt. On a write of 1, the mask of the **UARTRXINTR** interrupt is set. A write of 0 clears the mask.
3	DSRMIM	**nUARTDSR** modem interrupt mask. A read returns the current mask for the **UARTDSRINTR** interrupt. On a write of 1, the mask of the **UARTDSRINTR** interrupt is set. A write of 0 clears the mask.
2	DCDMIM	**nUARTDCD** modem interrupt mask. A read returns the current mask for the **UARTDCDINTR** interrupt. On a write of 1, the mask of the **UARTDCDINTR** interrupt is set. A write of 0 clears the mask.
1	CTSMIM	**nUARTCTS** modem interrupt mask. A read returns the current mask for the **UARTCTSINTR** interrupt. On a write of 1, the mask of the **UARTCTSINTR** interrupt is set. A write of 0 clears the mask.
0	RIMIM	**nUARTRI** modem interrupt mask. A read returns the current mask for the **UARTRIINTR** interrupt. On a write of 1, the mask of the **UARTRIINTR** interrupt is set. A write of 0 clears the mask.

PL190 Registers – Interrupt-related Registers

0x038	UARTIMSC	RW	0x000	11	*Interrupt Mask Set/Clear Register, UARTIMSC* on page 3-17
0x03C	UARTRIS	RO	0x00-	11	*Raw Interrupt Status Register, UARTRIS* on page 3-19
0x040	UARTMIS	RO	0x00-	11	*Masked Interrupt Status Register, UARTMIS* on page 3-20
0x044	UARTICR	WO	-	11	*Interrupt Clear Register, UARTICR* on page 3-21

PrimeCell UART (PL011) Technical Reference Manual

Raw/Masked Interrupt Status Registers (UARTRIS & UARTMIS)

Both registers have the same bit field as the UARTIMSC register.

The UARTRIS Register is the raw interrupt status register. It is ***read-only register***.

The UARTMIS Register is the masked interrupt status register. It is read-only register.
The status of the raw interrupts (UARTRIS) are masked by the enabled bits (UARTIMSC).

PL190 Registers – Interrupt-related Registers

0x038	UARTIMSC	RW	0x000	11	*Interrupt Mask Set/Clear Register, UARTIMSC* on page 3-17
0x03C	UARTRIS	RO	0x00-	11	*Raw Interrupt Status Register, UARTRIS* on page 3-19
0x040	UARTMIS	RO	0x00-	11	*Masked Interrupt Status Register, UARTMIS* on page 3-20
0x044	UARTICR	WO	-	11	*Interrupt Clear Register, UARTICR* on page 3-21

PrimeCell UART (PL011) Technical Reference Manual

Masked Interrupt Status Register (UARTICR)

The register has the same bit field as the UARTIMSC register.

The UARTICR register is a ***write-only register***.

On a write of 1 to a particular bit, the corresponding interrupt is cleared.
A write of 0 to a particular bit has no effect.

When is the UART requesting an interrupt?

3.2 Summary of registers

Table 3-1 lists the UART registers.

Table 3-1 UART register summary

Offset	Name	Type	Reset	Width	Description
0x034	UARTIFLS	RW	0x12	6	*Interrupt FIFO Level Select Register, UARTIFLS* on page 3-17

PrimeCell UART (PL011) Technical Reference Manual

Interrupt FIFO Level Select Register (UARTIFLS)

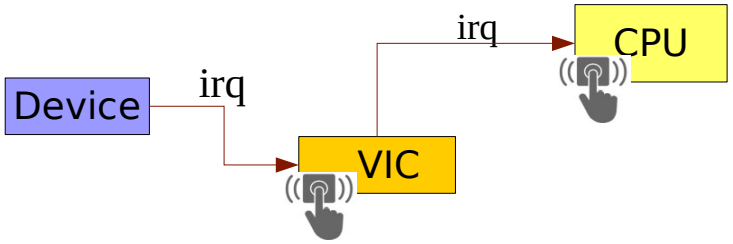
The interrupts are generated based on a transition ***through a level*** rather than being based on the level. That is, the interrupts are generated when the fill level progresses through the trigger level.

Enabling Interrupts

Enabling at the device…

Enabling at the VIC…

Enabling at the CPU (rather at the core level)



PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual

PL190 – Registers

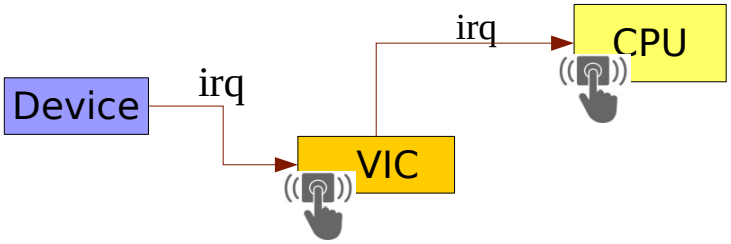
PrimeCell Vectored Interrupt Controller (PL190) Technical Reference Manual

Chapter 3 Programmer's Model

Table 3-1 VIC register summary

Register	Address offset	Type	Reset value	Description
VICIRQSTATUS	0x000	RO	0x00000000	See *IRQ Status Register* on page 3-6
VICFIQSTATUS	0x004	RO	0x00000000	See *FIQ Status Register* on page 3-6
VICRAWINTR	0x008	RO	-	See *Raw Interrupt Status Register* on page 3-6
VICINTSELECT	0x00C	R/W	0x00000000	See *Interrupt Select Register* on page 3-7
VICINTENABLE	0x010	R/W	0x00000000	See *Interrupt Enable Register* on page 3-7
VICINTENCLEAR	0x014	Write	-	See *Interrupt Enable Clear Register* on page 3-7

Summary with Enabled Interrupts



The instruction "halt" such as ***WFI*** or ***WFE*** for ARM processors

Vectored Interrupt Controller: (architected state per interrupt)

Trapping for the Interrupt Processing by the processor

FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC+4	Where the PC is the address of the instruction that was not executed because the FIQ or IRQ took priority
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC+4	

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

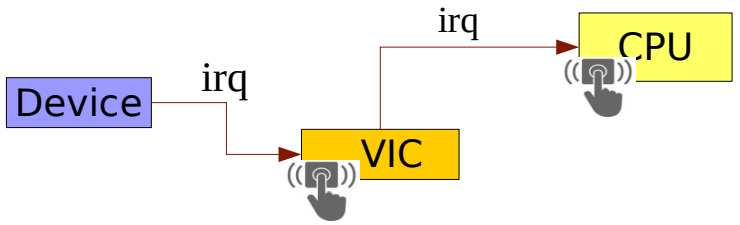
indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

```
0x00    br   _reset
0x04    br   _undef_inst
0x08    br   _soft_irq
0x0c    br   _prefetch_abort
0x10    br   _data_abort
0x14    br   _not_used
0x18    br   _isr
0x1c    br   _fisr
```

Enabled interrupts at devices…
Enabled interrupts at the VIC…
Enabled interrupts at the CPU...

Interrupt Service Routine – “Who is it?“

- Detect which interrupts are pending
  - Dialog with the VIC/GIC/PIC(1)
  - Using its mmio registers
    - Bit fields for raised interrupts
    - Bit fields for enabled/disabled interrupts
- Call the corresponding interrupt handlers
  - Indexed by the interrupt number
- *Acknowledge the raised interrupts*
  - Tells the device it has been serviced
  - If you do not acknowledged, the device will never raise its interrupt again...

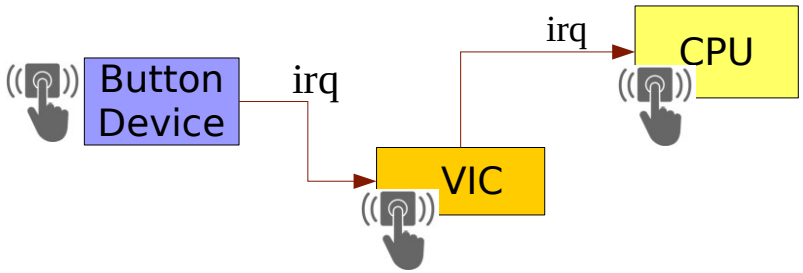



```
.global _isr
_isr:
    ...
    ; now call the interrupt_service_routine
    bl isr
    ...
```

```
struct handler {
  void (*callback)(void*);
  void* cookie;
};

struct handler handlers[MAX_NHANDLERS];

void isr() {
  uint32_t irqs = vic_load_irqs();
  for (uint32_t i=0 ; i<32 ; i++) {
    struct handler_t* handler;
    handler = &handlers[i];
    if (irqs & (1<<i))
      handler->callback(handler->cookie);
  }
  vic_ack_irqs(irqs);
  return;
}
```

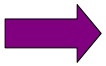
(1) GIC: Generic Interrupt Controller (ARM)
APIC: Advanced Programmable Interrupt Controller (Intel)

Coding with Interrupts



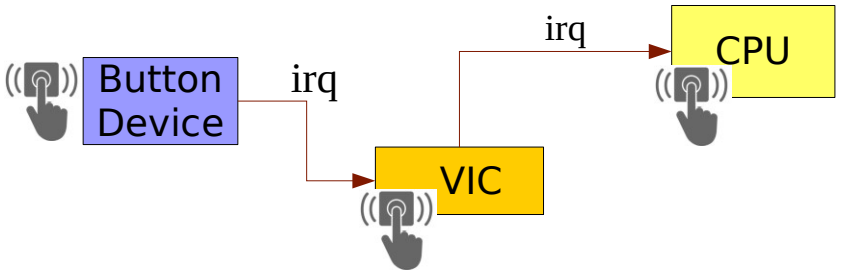

```
struct handler {
  void (*callback)(void*);
  void* cookie;
};
struct handler handlers[MAX_NHANDLERS];

void vic_enable_interrupt(uint32_t irqno,
                          void (*callback)(void*), void* cookie);
void vic_disable_interrupt(uint32_t irqno);

void core_enable_interrupts();
void core_disable_interrupts();
```

```
uint8_t button_get_status(void* bar);
void led_on(void* bar);
void led_off(void* bar);


void start() {

  button_init_regs(BUTTON_BAR);
  led_init_regs(LED_BAR);
  for (;;) {
     uint8_t status;
     status = button_get_status(BUTTON_BAR);
     if (status & 0x01)
       led_on(LED_BAR);
      else
       led_off(LED_BAR);
  }
}
```



```
void button_interrupt_handler(void* cookie) {
  uint8_t status;
  status = button_get_status(BUTTON_MMIO_BAR);
  if (status & 0x01)
    led_on(LED_MMIO_BAR);
  else
    led_off(LED_MMIO_BAR);
}

void _start() {
  init_interrupt_handlers();
  button_init_regs(BUTTON_MMIO_BAR);
  led_init_regs(LED_MMIO_BAR);
  vic_enable_interrupt(BUTTON_IRQ,
                       button_interrupt_handler,
                       null);
  button_enable_interrupt();
  core_enable_interrupts();

  for (;;) {
    halt();
  }
}
```

Summary – Hardware/Software Cooperation for Interrupts




```
0x00    ldr  pc, [pc,#0x18]
...
0x18    ldr  pc, [pc,#0x18]
0x1c    ldr  pc, [pc,#0x18]
0x20    .word _reset
...
0x38    .word _isr

_isr:
    ...
    bl isr
    ...
```

```
void button_interrupt_handler(void* cookie) {
  uint8_t status;
  status = button_get_status(BUTTON_MMIO_BAR);
  if (status & 0x01)
    led_on(LED_MMIO_BAR);
  else
    led_off(LED_MMIO_BAR);
}

void _start() {
  init_interrupt_handlers();
  button_init_regs(BUTTON_MMIO_BAR);
  led_init_regs(LED_MMIO_BAR);
  vic_enable_interrupt(BUTTON_IRQ,
                       button_interrupt_handler,
                       null);
  button_enable_interrupt();
  core_enable_interrupts();

  for (;;) {
    halt();
  }
}
```

```
struct handler {
  void (*callback)(void*);
  void* cookie;
};

struct handler handlers[MAX_NHANDLERS];

void isr() {
  uint32_t irqs = vic_load_irqs();
  for (uint32_t i=0 ; i<32 ; i++) {
    struct handler_t* handler;
    handler = &handlers[i];
    if (irqs & (1<<i))
      handler->callback(handler->cookie);
  }
  vic_ack_irqs(irqs);
  return;
}
```

Classwork – Homework

- Step1 – Simple Echo Console
  - Leverage receive interrupts (RX) on the UART
  - Keep transmit interrupts (TX) disable on the UART
  - Analysis of the potential dangers
- Step2 – Bytes or Characters
  - Serial lines transmit bytes..
  - Observe character encoding?
    - Arrows on the keyboard…
    - French accentuated letters…
  - Functions wrapping some special control sequences
    - Clear the screen: "\033[H\033[J"
    - Cursor at: "\033[**r**;**c**H" the arguments (r=row , c=column) given as digits, like ‘3’ or ‘12’
    - Cursor move: up, down, left, and right (user arrow key sequences)
    - Cursor: invisible "\033[?25l" , visible "\033[?25h"