Control systems and Computer Networks

Discrete Time and Interrupts

Dr Alun Moon

Lecture 02.2

Part I

Inputs

Time

- In the real world we have to deal with time.
- The CPU is driven by a clock signal

- From the CPU point of view we can think of time as being in discrete chunks.
- External clock is 50 MHz
- Cortex M4 core clock 120 MHz
- Clock period $\sim 8\,\mathrm{ns}$

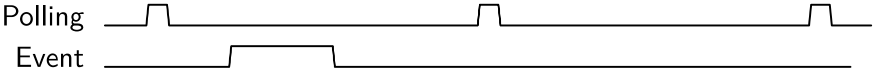
Polling

Consider the following code, polling the switch every 100 ms (10 times a second).

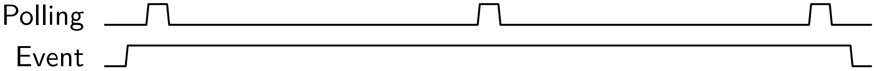
```
while(1) {
  if( ispressed(SW1) ) action();
  wait(0.1);
}
```

- The GPIO circuit looks at the switch for 10 ns
- We end up looking at the switch every 10 ns out of every 100 ms
- or for 0.00001% of the time.

We can miss important events...



- If the event we are watching for is smaller that the time between polls.
- We can fail to spot the event entirely

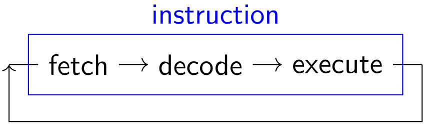


- For long events.
- When exactly is the button pressed?

Interrupts

Event driven programming

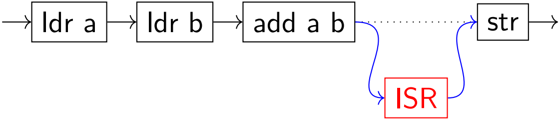
- Recall the *fetch-execute* cycle.



instruction

fetch → decode → execute

- Interrupts occur *asynchronously*



ldr a → ldr b → add a b → str

ISR

- When an Interrupt occurs (IRQ) the program jumps out of the normal flow, to the interrupt handler (ISR), then returns to the next instruction in the normal flow.

Digital Inputs

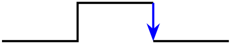
Edge triggered interrupts

- with a digital signal where do we raise an interrupt request?

- The easiest thing to do, is to detect changes in the signal

Rising Edge the signal goes from 0 to 1

Falling Edge the signal goes from 1 to 0



Interrupt Service Routines

ISRs cannot take parameters or return values

ISR prototype

```
void buttonISR( void );
```

Any data that needs to be passed between the ISR and the program needs to be done via *global variables*

ISRs need to be kept short. Remember the code is executing *between* other instructions.
Avoid slow operations such as reading and writing to the display or serial port.

Creating an Inturrupt handler

```
InterruptIn  left(SW2);
InterruptIn right(SW3);

left.rise(on);
right.fall(off);
while(1) /* GNDN */ ;
```

- Only some pins can generate interrupts
- an action can be attached to rising and falling edges
- Remember to check for logic inversions (pressed is a falling edge)
- Actions occur *independently* from the *main-loop*

Part II

Timers

Timeing

- For many applications we want something to happen periodically
- using loops and delays is problematic

```
while(1) {
    int sensor = ispressed(SW1);
    printf("button is %s presesed", sensor?"":"not" );
    wait(1);
}
```

- Timing depends on execution time of code.
  - difficult to predict
  - varies from loop to loop

Periodic Interrupt Timer

- We can generate interrupts from a hardware timer
- these can be set at a particular period
- an IRQ is generated an each period.
- Accurate precise times
  - lower resolution – the system clock $\sim 8\,\mathrm{ns}$
  - upper bound – when the counter rolls over $34\,\mathrm{s}$ for 32 bits
  - for 4 chained timers $2.7 \times 10^{30}\,\mathrm{s}$ (8×10^{22} a or 6×10^{12} universe)

Using a PIT

Soft timers

- Once we have a periodic *tick* we can do interresting things
- the ISR can count *tick*s

On tick

0 LED on

2 LED off

5 reset tick to 0

Turns the LED on for 2/5 of the time

MBed

The MBed library uses a single soft-timer to handle PIT interrupts

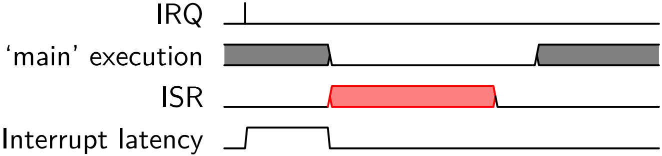
```
Ticker pit;
pit.attach(flash,0.5);
while(1);
```

- Attaches the `flash` function to be called every 0.5 s
- Happens independently to main-loop
- Concurrency! (without the messing about with OS)
- Library supports any number of PIT interrupts

Interrupt timing

The fine details

- There is a delay between the IRQ being raised and the ISR starting.



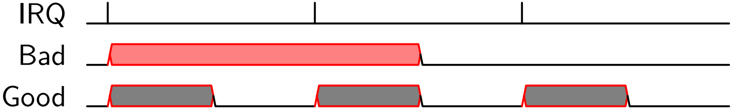
IRQ

‘main’ execution

ISR

Interrupt latency

- Remember not to make ISRs too long



IRQ

Bad

Good