Interrupt Handling

Philippos Papaphilippou & Klaus-Peter Zauner

COMP2323: Computer Systems II

University of Southampton

By now you should. . .

0. have a logbook, a PiPIco2W, and a working tool-chain.
1. be able to cross-compile a C program for RSIC-V.
2. be able to program a PiPico2W with your own binary.
3. be able to show that your code is running on the device.

Side Note: LED User Interface

Pins are precious in small devices, an LED is a luxury and often as good as it gets for an embedded user interface.

- How many different states can you clearly indicate with a single one-colour LED?
- What else could you do with it?

LEDs are also quite power hungry compared to a small microcontroller—a low duty cycle is a good idea.

Side Note: LED User Interface

- You can use Easing functions, e.g. [blog.febucci.com]] with PWM brightness control, and chain them up
- You can overlay a fast serial protocol and read it with and an app through a mobile phone camera

Side Note: LED User Interface

- You can use Easing functions, e.g. [blog.febucci.com]] with PWM brightness control, and chain them up
- You can overlay a fast serial protocol and read it with and an app through a mobile phone camera

Interrupt Handling

Basic I/O Methods

Programmed I/O (Polling)

- CPU sits in a tight loop until input is available or output can be accepted
- Occupies CPU ("busy waiting")
- Very fast reaction possible

Interrupt-driven I/O

- Hardware signal can change program flow
- CPU can do other work (or sleep) unless I/O is possible

Other I/O methods like Direct Memory Access (DMA) or dedicated I/O computers (e.g. a graphics card) use interupts to communicate with the CPU.

Basic I/O Methods

Programmed I/O (Polling)

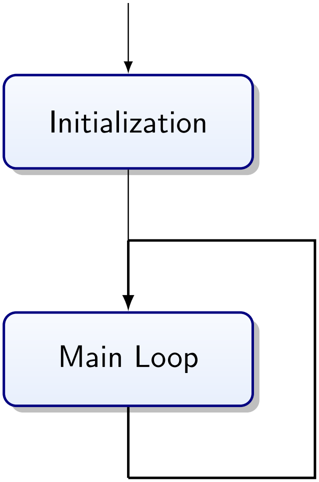
- CPU sits in a tight loop until input is available or output can be accepted
- Occupies CPU (”busy waiting”)
- Very fast reaction possible

Interrupt-driven I/O

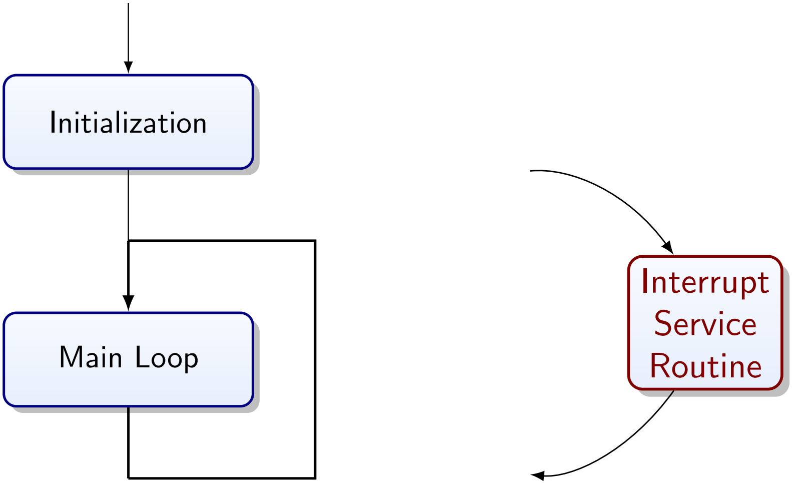
- Hardware signal can change program flow
- CPU can do other work (or sleep) unless I/O is possible

Other I/O methods like Direct Memory Access (DMA) or dedicated I/O computers (e.g. a graphics card) use interupts to communicate with the CPU.

Program Flow



Program Flow



Interrupts

The solution to overcome the inefficiency of programmed I/O is a possibility to interrupt the CPU when I/O devices are ready to receive or deliver data.

Important: the CPU needs to be able to continue where it was interrupted after the interrupt has been dealt with

- State of the interrupted process needs to be preserved

Exception: if interrupt should abort execution because of an irrecoverable fault

Interrupts

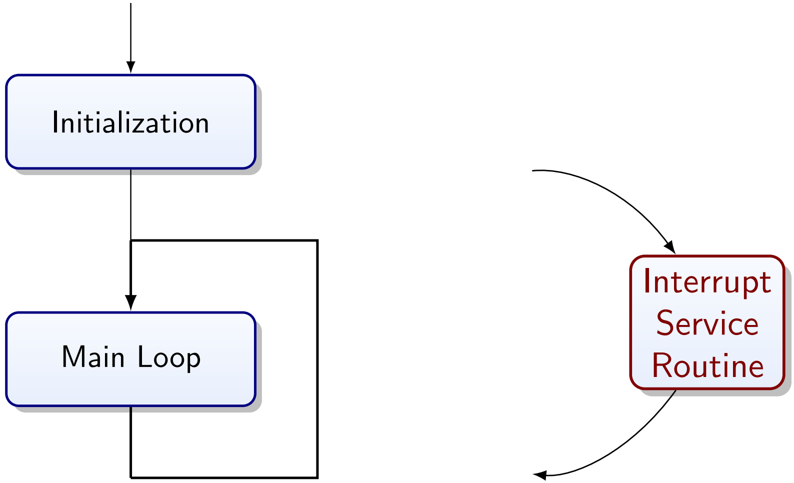
The solution to overcome the inefficiency of programmed I/O is a possibility to interrupt the CPU when I/O devices are ready to receive or deliver data.

Important: the CPU needs to be able to continue where it was interrupted after the interrupt has been dealt with

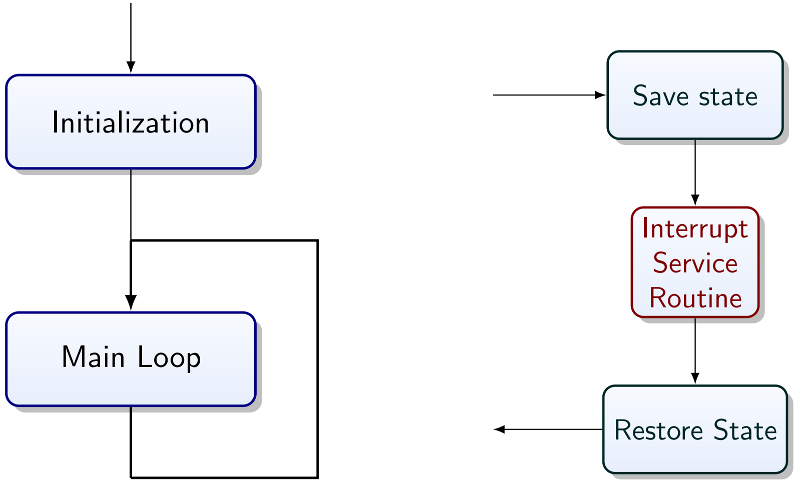
- State of the interrupted process needs to be preserved

Exception: if interrupt should abort execution because of an irrecoverable fault

Program Flow



Program Flow



“Precise Interrupt”

1. State of program counter (PC) is preserved
2. Everything before PC has been fully executed
3. Nothing beyond the PC has been started
4. Execution state of instruction at the PC is known

What happens when an Interrupt arrives?

- Processor completes current instruction
- Processor acknowledges interrupt
- Hardware saves some state:
  - Program counter (PC)
  - Process status word (PSW)
- PC register is loaded with value from interrupt vector table

$\longrightarrow$ Control now handed to software

What happens when an Interrupt arrives?

- Software disables interrupts
- Saves additional state
  - Registers $\rightarrow$ Stack
- May reenable interrupts
  - Easier if not enabled
- Services the interrupt $\rightarrow$ ISR
- Restore state (Software)
- Enable interrupts
- Hardware: restores PSW and PC

Interrupt Service Routines (ISR)

Procedure that is executed when the interrupt occurs and that handles the interrupt.

Two important rules:

1. Keep them fast
   - avoid loops
   - avoid heavy instructions → no `printf()`
   - should not block → no `scanf()`
2. Keep them simple
   - debugging ISRs is hard

Interrupt Service Routines (ISR)

Procedure that is executed when the interrupt occurs and that handles the interrupt.

Two important rules:

1. Keep them fast
   - avoid loops
   - avoid heavy instructions → no `printf()`
   - should not block → no `scanf()`
2. Keep them simple
   - debugging ISRs is hard

Interrupt Service Routines (ISR)

Procedure that is executed when the interrupt occurs and that handles the interrupt.

Two important rules:

1. **Keep them fast**
   - avoid loops
   - avoid heavy instructions → no `printf()`
   - should not block → no `scanf()`
2. **Keep them simple**
   - debugging ISRs is hard

Why fast?

If they are fast you can usually block all interrupts:

- Life is simpler if you do not need to make your ISRs interruptable
- Stack size bounds are simpler to establish
- No need for reentrant ISRs

Latency

- How long does it take until the CPU can respond?
- Is this delay deterministic?

Deterministic latency may be important in real-time applications: e.g., human operators and also control algorithms can adapt to deterministic latency, but struggle with random delays in control lines.

Latency

- How long does it take until the CPU can respond?
- Is this delay deterministic?

Deterministic latency may be important in real-time applications: e.g., human operators and also control algorithms can adapt to deterministic latency, but struggle with random delays in control lines.

Maximum Latency

We have latency due to hardware:

- Current instruction is completed
- Hardware support to save state

We also have latency due to software:

- Software to save state
- Maximum length of critical sections that disable interrupts

Maximum Latency

We have latency due to hardware:

- Current instruction is completed
- Hardware support to save state

We also have latency due to software:

- Software to save state
- Maximum length of critical sections that disable interrupts

Jitter

- Instructions cannot be interrupted
  - c.f. DMA/CPU halting
- Some instructions take more than one clock cycle
- The response time depends on the instruction executed when the interrupt arrives

How to keep ISRs fast and simple?

- Keep Interrupts off during ISR

Advantage:

- No worries about stack depth
- No overhead for reentrant code

Disadvantage:

- Latency
- Lost interrupts

$\longrightarrow$ ISR needs to be short/fast (always bounded!)

How to keep ISRs fast and simple?

- Move the data that needs processing to some buffer
- Set a global flag $\longrightarrow$ `volatile unit8_t`
- Return immediately
- Check the flag in the main loop and do the work there

ISRs in C

- The C language unfortunately has no support for ISRs
- Declaring ISRs is compiler specific
- See the avr-libc documentation

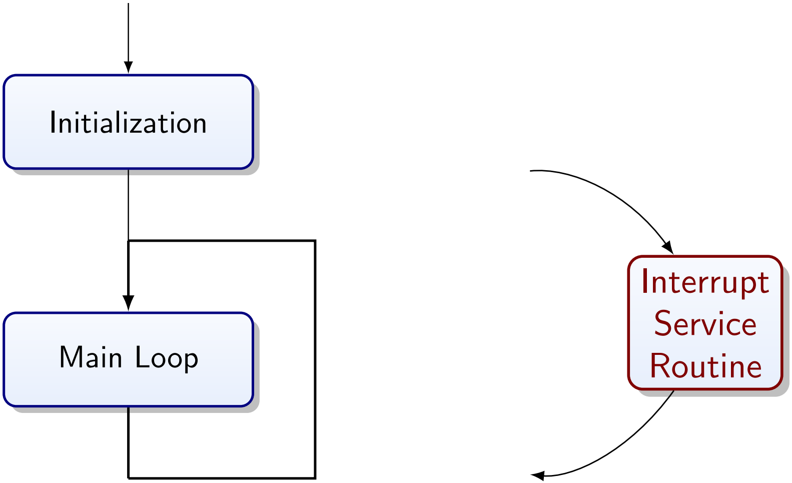
ISRs in C

Instructions are not interrupted, but. . .

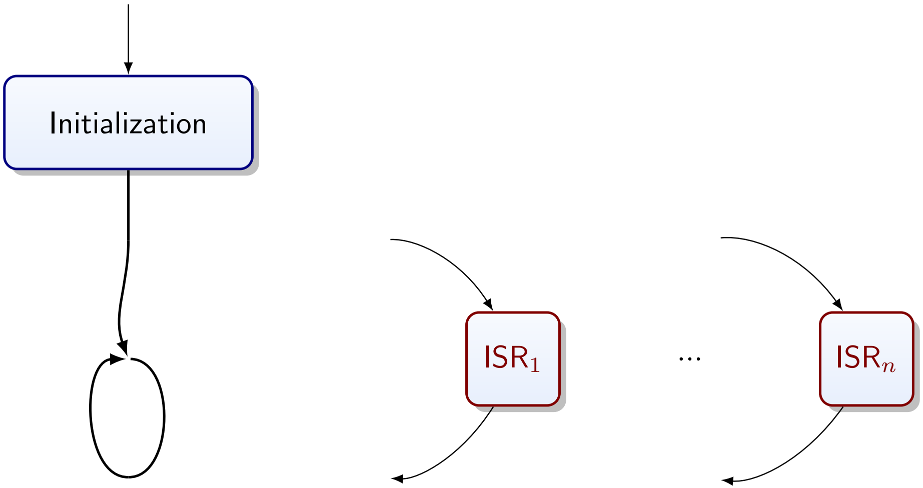
- C statements are typically compiled into multiple CPU instructions
- C statements can be interrupted!

Event Driven Programming

Program Flow

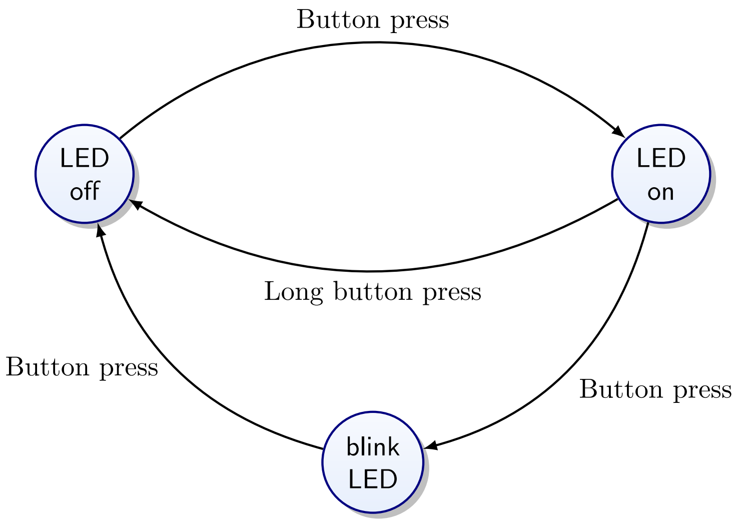


Program Flow



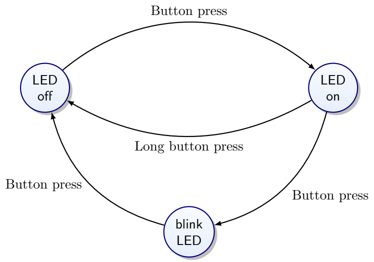
The main loop may be empty or a sleep command.

Bicycle light



Almost certainly implemented on a microcontroller.

State Machines



- Moore machine:
  → output depends only on state
- Forever loop
- Events drive transitions

Internal and External Events

In general, events can be internal, e.g.

- a timer overflow
- completion of ADC conversion

or external, e.g.

- analouge comparator exceeds treshold
- keyboard input
- display refresh cycle

Events

- Maximum arrival rate?
  - physical limit?
  - scheduler?
- Deadline for servicing?
  - Cost if missed? $\longrightarrow$ hard to debug
  - What part is time sensitive?
- Longest time interrupts are disabled?
- Impact on other realtime code?

Interrupt Vectors & Interrupt Service Routine

- Different interrupt sources $\longleftarrow$ Events
- Each is associated with an ISR:

Interrupt Service Routine (ISR)

Procedure that is executed when the interrupt occurs and that handles the interrupt.

There needs to be an ISR for every interrupt source that is enabled. For the processor to know where to branch to there is a table at the start of the program memory with the interrupt vector the address of the ISR for each source.

Vector	*Address*	*Source*	*Interrupt definition*
1	$0000	RESET	Reset pin, Power-on -, Brown-out -, Watchdog -, JTAG reset
2	$0002	INT0	External Interrupt Request 0
⋮			
9	$0010	INT7	External Interrupt Request 7
10	$0012	PCINT0	Pin Change Interrupt Request 0
⋮			
13	$0018	WDT	Watchdog Time-out Interrupt
⋮			
16	$001E	TIMER2	OVF Timer/Counter2 Overflow

For the full list of all 38 vectors see DS p68.

ISR: Things to watch out for...

Variables

- A variable that is used in the ISR and the main program needs to be declared `volatile`
  - this lets the compiler know that it can not be cached in a register
- Multi-byte variables:
  $\rightarrow$ access atomically outside ISR

`volatile` $\rightarrow$ slow

- `volatile` will turn off all optimization for this variable

`volatile` → slow

- `volatile` will turn off all optimization for this variable
- If a volatile variable is used a lot in the ISR, copy it into a local variable
  - this will work if the ISR does keep interrupts disabled

ISR: Things to watch out for. . .

Registers

- Operations on registers that are used by the ISR and the main program need to be atomic

atomic = completes without interruption

ISR Implementation

1. Register ISR
2. Enable interrupt at device level
3. Globally enable Interrupts

Acknowledgments

These slides were produced with Teodor Nistor's **beamr**:

https://teonistor.github.io/beamr/