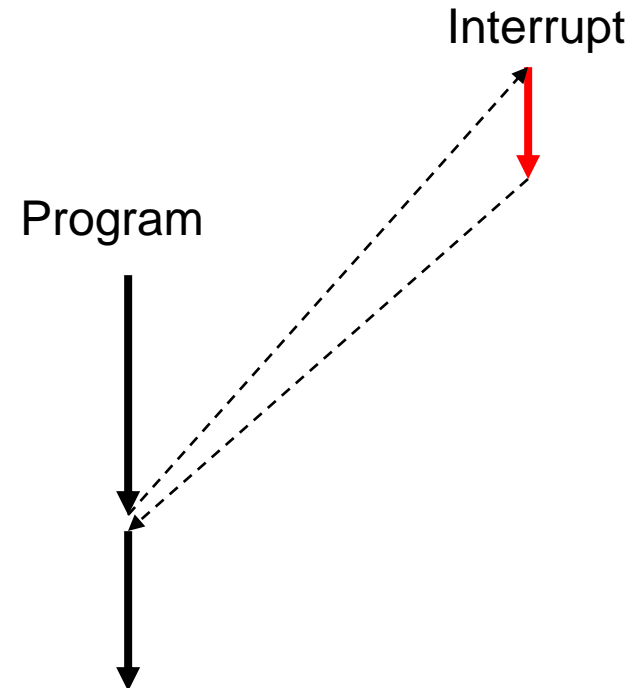


Interrupts

- Interrupts are a mechanism which makes processor to respond quickly to external (or internal) stimulus
 - For example when IO-device switches to ready-state
- Current execution is interrupted and processor jumps to interrupt handler code. After handler is executed processor resumes execution at the address where interrupt occurred
- Interrupt is "invisible". The program that is interrupted is not interfered by interrupt handler (except for the delay caused by interrupt handler execution)



Interrupt is hardware coerced subroutine

- Ordinary subroutine needs only to save registers that calling convention defines as callee saved
 - C/C++ compiler takes care of saving registers for you
- When you call an ordinary subroutine you can prepare for subroutine call by saving important registers that may be modified by the subroutine
- Interrupts are triggered independently of the program execution thus interrupt handler needs to ensure that interrupted program is not disturbed
 - Interrupt handler needs to save **all** registers that it modifies
 - **Flags** must be preserved
 - For example interrupt can occur between comparison instruction and conditional branch
 - When an interrupt routine is written in C/C++ you must tell the compiler that the routine is an interrupt handler

```
GCC: void handler() __attribute__((interrupt ("IRQ")));
```
- Our ARM cortex-M3 is an exception (see next slide)

ARM Cortex-M3 interrupt handling

- ARM Cortex-M processors target low-end and mid-range embedded systems
- ARM has implemented some extra hardware measures to simplify the way interrupt handlers are written
 - ARM has laid out recommended register usage convention that defines which registers need to be saved by a called function
 - ARM has implemented hardware that saves the other registers automatically before interrupt handler is entered
 - If compiler follows ARM register usage convention interrupt handler can be an ordinary function (no special attributes are needed)
 - GCC follows ARM register usage convention

AAPCS register usage

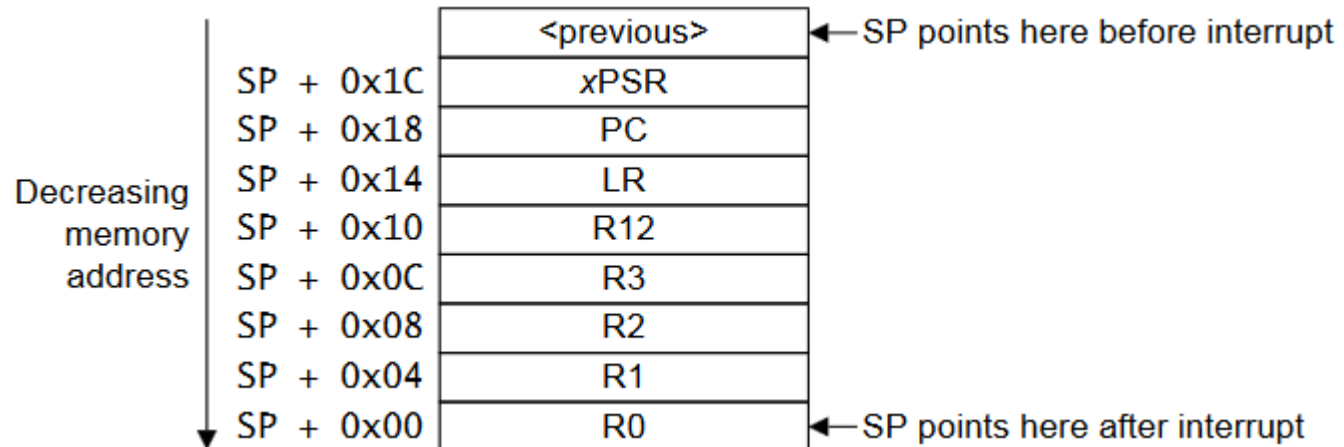
- ARM standard requires that when a function is called the function must preserve the contents of registers R4 – R11.
 - A function written in C/C++ will save and restore them if they are used by the function

AAPCS compliant compiler
will preserve these registers

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Automatic register stacking

- When an interrupt occurs the processor automatically pushes some of the register on the stack
 - Since compiler will preserve R4 – R11 they don't need to be automatically saved
 - Only the registers that compiler is free to use are automatically pushed on the stack



Interrupt handler

- Processors have multiple interrupt sources
- Interrupt handler needs to find out which of the sources caused the interrupt and then handle the interrupt
 - Handling means that interrupt is acknowledged and the action associated with interrupt is performed (for example data transfer)
 - Interrupt is acknowledged by
 - Bus activity
 - Read or write to a special address (for example status register)
 - External signal
 - Some sources need no acknowledge at all. For example systick timer needs no acknowledge
- Most modern microcontrollers use interrupt vectors
 - Each type of interrupt has a dedicated memory location where the interrupt handler address is stored. Hardware automatically calls appropriate handler.

Cortex-M3 interrupt vectors

- Interrupt vector is the address to which hardware jumps when an interrupt occurs
- Compiler/linker fills in the vector table with addresses of the interrupt handlers
 - LPCXpresso uses a system where vectors are identified with dedicate names
 - For example SysTick-interrupt handler must be called SysTick_Handler in order to be placed in right position in the vector table
 - C++ uses namespaces function verloading. The internal name of a function (for linker) is formed from namespace, parameter types, and the function name. Interrupt handler must be declared wrapped insize "extern "C" {} to force the linker name to be the same as the name in source file

number	Offset	Vector
0x0040+4n		IRQn
		...
		...
0x004C		IRQ2
0x0048		IRQ1
0x0044		IRQ0
0x0040		Systick
0x003C		PendSV
0x0038		Reserved
		Reserved for Debug
		SVCall
0x002C		Reserved
		Usage fault
0x0018		Bus fault
0x0014		Memory management fault
0x0010		Hard fault
0x000C		NMI
0x0008		Reset
0x0004		Initial SP value
0x0000		

Interrupt handler

- Save processor state
 - Return address (some processors do this automatically)
 - General purpose registers
 - Flags
 - Special registers (processor dependent)
 - Stack pointer
- Acknowledge interrupt
- Execute required action (read/write data)
- Restore processor state
- Return to location where interrupt occurred

Processor state

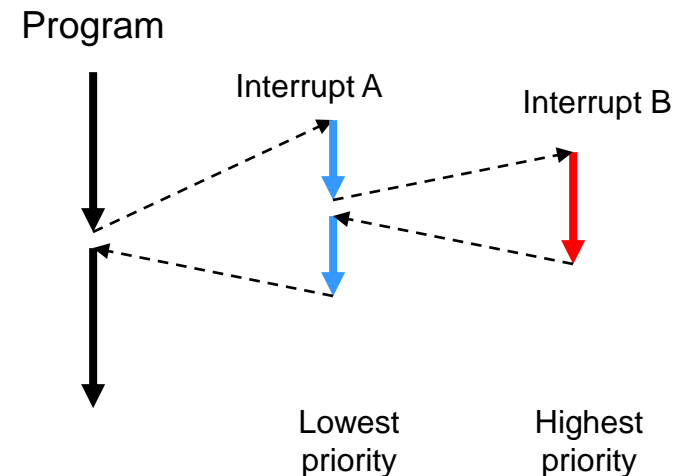
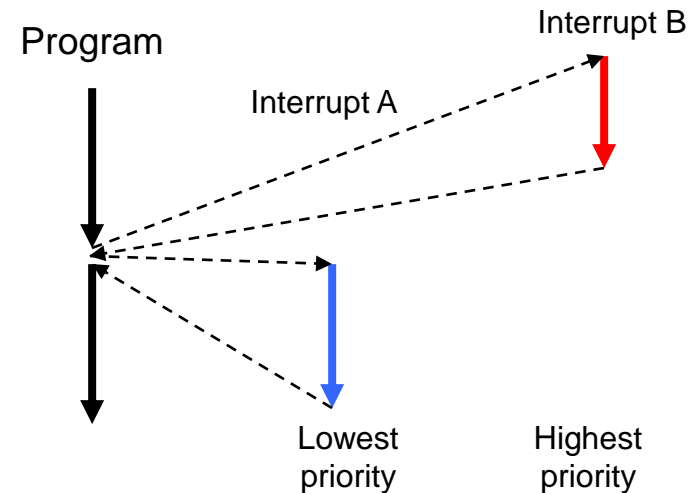
- Processor state is saved to stack
 - The interrupt handler needs not to know the absolute address
- Processor saves return address automatically
 - Some processors save return address automatically to stack
 - Others save return address to a special register and user must push return address to stack
- Special instruction for returning from interrupt
 - If processor state is modified in interrupt activation it must be restored at the same time with return from interrupt handler

Interrupt categories

- Device interrupts (hardware interrupts)
 - Occurs asynchronously
 - From external or integrated peripheral device (UART, AD-converter, NIC, etc.)
- Interrupts caused by software
 - Current instruction causes interrupt
 - Software interrupt (explicit request)
 - Others:
 - Illegal instruction
 - Illegal memory access
 - Page fault
 - Etc.

Interrupt priorities

- Interrupts have priorities which determine the order of handlers if two interrupts occur simultaneously
 - Higher priority is executed before lower priority
- Interrupts can also be nested
 - Higher priority interrupt can interrupt lower priority handler
 - Requires larger stack
- Priorities are assigned by programmer or hardware designer
 - Some processors have fixed priorities that can not be changed by the software



Interrupts can be disabled

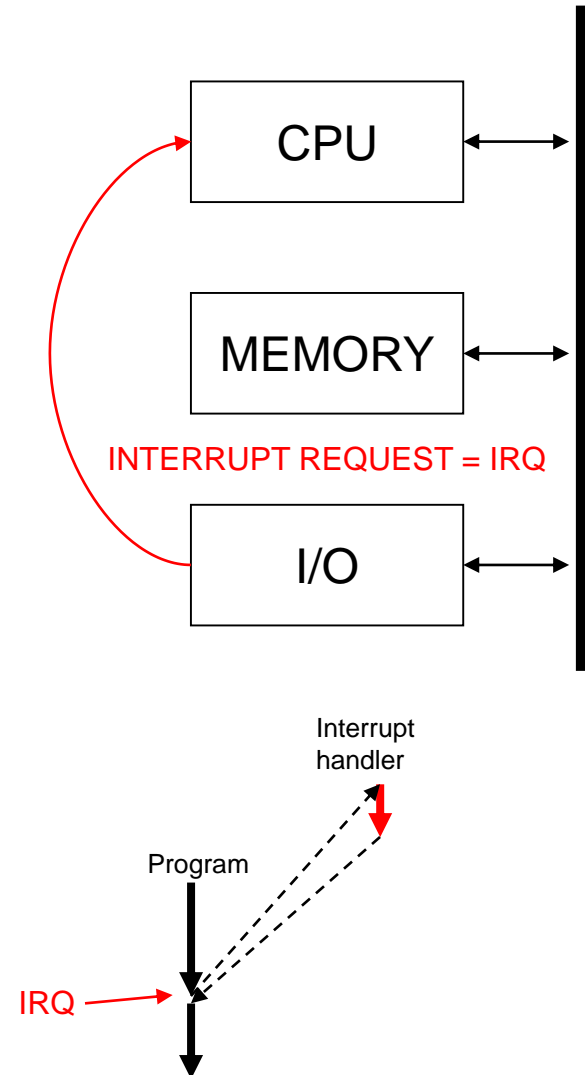
- Some parts of program may not be interrupted
 - Such a part is called a critical section
 - Interrupts must be disabled during execution of critical section
 - Increases interrupt response time = time between interrupt signal and handler execution
 - Minimize the time that interrupts are disabled – make your critical section as short as possible

Interrupts and privileges

- If a processor has at least two privilege levels (user mode and privileged mode) the interrupts handler runs in the privileged mode
 - User mode access to hardware resources is restricted. For example processor may prevent access to IO-devices, memory management and prevent user from changing privilege level
 - In privileged mode there are no access restrictions
 - Operating system kernels run in privileged mode
 - Whether user programs run in privileged mode depends on the operating system. Unix style operating systems restrict user privileges while many RTOSes allow programs to run in privileged mode as well
 - Operating system calls can be implemented with software interrupts which switches to privileged mode automatically on OS calls

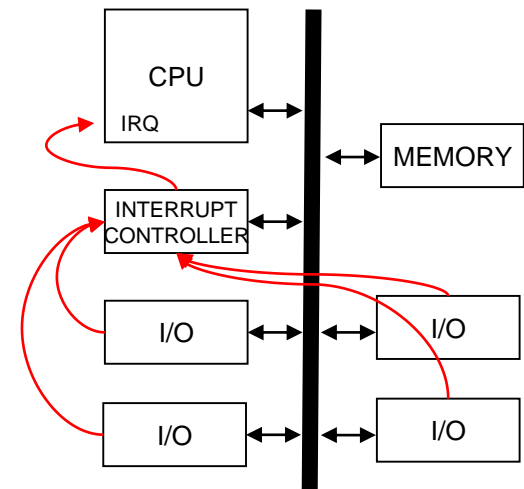
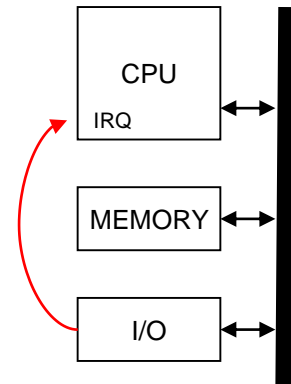
Interrupt driven IO

- IO device is connected to the interrupt input of the CPU
- When device needs attention it asserts the interrupt signal
- Processor interrupts the currently executing program and jumps to interrupt handler
- Interrupt handler executes the actions that the device requires and acknowledges the interrupt
 - For example copies the data from IO device
 - If further processing is needed that is done in the main program – the handler and main program need to exchange some information



Interrupt driven IO

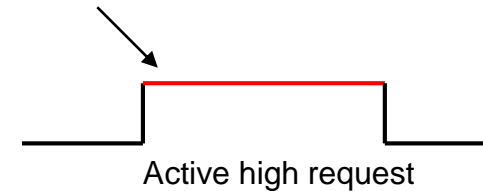
- Interrupt controller receives interrupt requests and passes them to the CPU
 - Can handle several different types of interrupt signaling
 - Prioritizes the interrupts
 - Keeps track of pending interrupts
 - Has storage for interrupt vectors (handler functions addresses)
- In a modern microcontroller the interrupt controller is integrated to the CPU



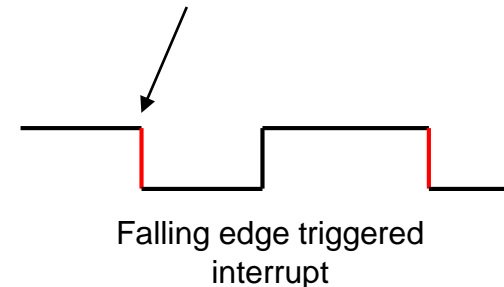
Interrupt signal types

- Level sensitive interrupt
 - Signal level indicates interrupt request
 - Request is active as long as signal is active
 - Interrupt handler must acknowledge interrupt to deactivate interrupt request
- Edge sensitive interrupt
 - Change of signal indicates interrupt request
- Type of interrupt signal is configured by the programmer
 - Can (typically) be configured for each interrupt source
 - For external signals

Interrupt is active as long as signal is high



Signal must go high before new interrupt can occur



Interrupt driven IO

- USB keyboard is attached to a microcontroller that runs at 72 MHz
 - USB HID transfer rate is max 64 kbps (8 kilobytes per second)
 - One interrupt takes 300 clock cycles and transfers one character
 - If user types twelve eighth character words per minute (~100 chars per minute) then the interrupt is executed only 100 times

$$\frac{100 \times 300}{60 \times 72 \times 10^6} \approx 0,0000694 \%$$

- Benefits of interrupt driven IO
 - Fast response time
 - Processor resources are used only when needed

How to communicate with an interrupt handler

- If an operating system is used then OS primitives are used for communication
 - A typical RTOS provides signals and queues for communication
- If there is no operating system then a shared memory region (variables, structure or object)
 - The simplest way is to set a flag (= global variable) in the interrupt handler and when the main program sees the set flag it does its part of the processing
 - Makes sense only in a small system where timings can be fully analyzed
 - Usually ISR puts data/events in a queue
 - For example received characters go into a buffer where they can be retrieved and processed by the main program
 - An operating system uses similar mechanisms but hides them behind the OS API

Critical section

- Access to a shared memory region must be an atomic operation
 - While main program is accessing the shared memory ISR is not allowed to change any values → critical section
- Think of a case where ISR and main program share a buffer for data and a counter to indicate the amount of data in the buffer

Main program:

```
if(counter > 0) {
    data = get_value_from_buffer();
    counter = counter - 1;
}
```

Diagram showing the execution of the main program's counter decrement operation. The counter is initially 3 (green hexagon). An arrow points to the assignment statement, which is underlined. The counter is then decremented to 1 (green hexagon).

What happens if ISR executes after (1) but before the assignment?

2

ISR (interrupt handler):

```
data = read_from_io_device();
put_value_in_buffer(data);
counter = counter + 1;
```

Incrementing/decrementing a variable is a **read-modify-write** sequence which must be atomic!

Critical section

- The simplest way to implement a critical section that makes shared memory access atomic is to disable interrupts during the execution of the critical section
 - ARM provides CMSIS software library that has functions that are needed to enable and disable interrupts
 - The functions to use are `__get_PRIMASK()`, `__disable_irq()` and `__enable_irq()`
- On Cortex-M3 it also possible to mask (=prevent execution) interrupts using BASEPRI register. BASEPRI can be used to mask interrupts based on their priorities instead of masking all interrupts

Functions to enter and exit critical section ...

```
// returns the interrupt enable state before entering critical section
```

```
bool enter_critical(void)  
{  
    uint32_t pm = __get_PRIMASK();  
    __disable_irq();  
    return (pm & 1) == 0;  
}
```

```
// restore interrupt enable state
```

```
void leave_critical(bool enable)  
{  
    if(enable) __enable_irq();  
}
```

... and how to use them

```
bool irq;
```

```
irq = enter_critical();
```

```
if(counter > 0) {  
    data = get_value_from_buffer();  
    counter = counter - 1;  
}
```

Interrupts are disabled while this is executed.

```
leave_critical(irq);
```

- If you use critical sections keep them as short as possible
- Disabling interrupts increases interrupt latency (reaction time)

The C++ way of handling critical section

```
class Imutex {  
public:  
    Imutex();  
    ~Imutex();  
    void lock();  
    void unlock();  
private:  
    bool enable;  
};
```

```
#include "chip.h"  
  
#include "Imutex.h"  
  
Imutex::Imutex() : enable(false)  
{  
}  
  
Imutex::~Imutex()  
{  
}  
  
void Imutex::lock()  
{  
    enable = (__get_PRIMASK() & 1) == 0;  
    __disable_irq();  
}  
  
void Imutex::unlock()  
{  
    if(enable) __enable_irq();  
}
```

The C++ way of handling critical section

```
// manual use of critical section

// create critical section object
Imutex guard;

// do something here...

guard.lock(); // disable interrupts
// do something here and do it quickly
// because this is inside critical section
guard.unlock(); // enable/restore interrupts
```

```
// automatic critical section with lock_guard
#include <mutex>

class example {
    void example_function()
    {
        // interrupts are now automatically
        // disabled at the beginning
        std::lock_guard<Imutex> lock(guard);
        // and enabled/restored at the end
    }

private:
    Imutex guard;
};

void second_example()
{
    Imutex guard;
    std::lock_guard<Imutex> lock(guard);
    // this function is completely inside
    critical section
}
```


Safe Read-modify-write with atomic template

- A critical section is not needed if the RMW-sequence is performed on atomic integer types
- `std::atomic` template provides interrupt (and multithread and multicore) safe way of modifying integer values (since C++11)

- Include `<atomic>`

```
volatile std::atomic_int counter;
extern "C" {
void SysTick_Handler(void)
{
    Board_LED_Toggle(0);
    Board_LED_Toggle(1);
    if(counter > 0) --counter;
}
}

int main() {
    // do something here...
    counter = 10;
    while(counter > 0) __WFI();
}
```

About interrupt latency

- In certain cases the use of critical sections can be avoided all together
- A classic example is a ring buffer with head and tail pointers
 - Producer modifies only head pointer and reads tail pointer
 - Consumer modifies only tail pointer and reads head pointer
 - This way there is only one writer for each pointer which means that writes can never be corrupted by interrupts
- Multi-core atomic memory access is a completely different story
 - There are means for that in the ARM architecture but that is out of the scope of this course

Interrupt latency

- Interrupt latency has two components: the execution time of higher priority interrupts and the time that interrupts are disabled
- How to estimate the worst case latency?
 - Assume that all higher priority interrupts are triggered at the same time
→ sum up their execution times
 - Add the longest time that interrupts are disabled anywhere in the program (a single case, not a sum of disable times)
 - Analyze if your ISR can starve – if higher priority interrupts are triggered at such a high rate that the ISR in question never gets to execute you are in big trouble
- In most cases the worst case will not occur very often – but in a time critical system you must be prepared for it

Make ISR as quick as possible

- The ISR should be as short as possible and do only time critical things that can't be delayed to a later time
- Three things that ISR must avoid
 - Wait for something – an ISR that needs to wait indicates incomplete analysis of system behaviour
 - Allocate or free memory (new/delete or malloc/free)
 - Call library functions (print, file handling etc.)
- The golden rule of ISR programming: Get in – get out (quickly)

```
void pulse_isr(void)
{
    volatile int channels;

    /* read which channel(s) caused interrupt */
    channels = LPC21_T0IR;
    /* clear interrupt from timer HW */
    LPC21_T0IR = channels;

    pulses++;

    if(--precounter == 0) {
        precounter = prescaler;
        old_length = pulse_length;
        pulse_length = LPC21_T0CR0;
    }
}
```


```
void uart_handler(int uart_nr)
{
    int interrupt;
    int status;
    UART_PARAMS *u;
    UART_STATE *s;

    u = &uart[uart_nr];
    s = &state[uart_nr];

    do {
        if (interrupt == *u->iir;
        if ((interrupt & 0x01) == 0) {
            status = *u->lsr;
            if (interrupt & INTR_XMT_EMPTY) {
                if (status & LSR_THRE) {
                    uart_transmit(u, s);
                }
            }
            else {
                while (status & LSR_RECEIVE) {
                    if (status & LSR_RECEIVE_ERROR) {
                        uart_error(u, s);
                    }
                    else {
                        uart_receive(u, s);
                    }
                    status = *u->lsr;
                }
            }
        }
    } while ((interrupt & 0x01) == 0);
}
```

Wait for interrupt

- Many embedded systems are event driven – most of the processing is triggered by an event
 - In a properly designed systems events are triggered by interrupts
 - IO devices, timers, etc.
- Waiting for something requires that you have a loop that repeats until the event occurs
 - Since events are triggered by interrupts there is no need to do any checking between the interrupts
 - You can put the processor to idle/sleep state and wake up when an interrupt occurs – then check and processes events and when you are done put the processor back to sleep again
- ARM has a special instruction that puts processor to sleep until an interrupt occurs. The instruction is exposed to the user by the driver library as `__WFI()` function

Sleep while waiting 

```
void Sleep(int ms)
{
    counter = ms;
    while(counter > 0) {
        __WFI();
    }
}
```

Event handler

- A practical approach to event handling is to put events into a queue. Event dispatcher then reads the queue and processes them in order
- Queuing preserves event order even if the processing would be delayed

