

# M2

---

## Real-Time Operating Systems

---

Unlike general-purpose operating systems, a real-time operating system (RTOS) is designed for embedded applications where time-sensitive operation is required. It provides a predictable (deterministic) execution pattern, allowing tasks to be completed before a deadline. This lab looks at hand-coded 'stand-alone' alternatives, and the operation of an RTOS on a resource-constrained microcontroller.



Image from <http://www.freertos.org/logo.jpg>

---

**Schedule**

---

Preparation time : 3 hours

Lab time : 3 hours

---

**Items provided**

---

Tools : Standard toolkit

Components : None

Equipment : Digital test bed, lift, lift connection module (on breadboard with ATmega162 processor), 5V FTDI cable, oscilloscope

Software : WinAVR

---

**Items to bring**

---

Essentials. A full list is available on the Laboratory website at <https://secure.ecs.soton.ac.uk/notes/ellabs/databook/essentials/>

**Before** you come to the lab, it is essential that you read through this document and complete *all* of the preparation work in section 2. If possible, prepare for the lab with your usual lab partner. Only preparation which is recorded in your laboratory logbook will contribute towards your mark for this exercise. There is no objection to several students working together on preparation, as long as all understand the results of that work. Before starting your preparation, read through all sections of these notes so that you are fully aware of what you will have to do in the lab.

**Academic Integrity** – *If you undertake the preparation jointly with other students, it is important that you acknowledge this fact in your logbook. Similarly, you may want to use sources from the internet or books to help answer some of the questions. Again, record any sources in your logbook.*

---

**Revision History**

---

March 16, 2014

Alex S. Weddell (asw)

Updated from previous M2 (by jnc)

## 1 Aims, Learning Outcomes and Outline

This laboratory exercise aims to allow you to:

- Experience the problems with using super-loops to control program operation
- Explore the functionality delivered by real-time operating systems
- Implement real-time programs using a real-time operating system

Having successfully completed the lab, you will be able to:

- Explain the difference between super-loop programs and operating system execution
- Implement a time-sensitive system using a real-time operating system

This lab follows on from the first-year lab on interrupts. Using AVR processors, we will look at how to deliver time-based functionality for systems in two ways. The first way is to use a super-loop, building all the system's functionality into an infinite loop (a super-loop). The second method is to use a real-time operating system, which can run several tasks simultaneously and manage their timings. You will learn about the benefits and drawbacks of each approach. At the end of the exercise, you will be able to use a real-time operating system to control the operation of a lift. This is applicable to embedded systems, where time-sensitive operation is required.

Additional resources for this exercise are available at <http://www.ecs.soton.ac.uk/notes/ellabs/2/m2>

---




## 2 Preparation

Read through the course handbook statement on safety and safe working practices, and your copy of the standard operating procedure. Make sure that you understand how to work safely. Read through this document so you are aware of what you will be expected to do in the lab.



### 2.1 Interrupts

This laboratory exercise is based on the ATmega162 microcontroller. It is from the same family as the ATmega644P microcontroller you have used in the first year with the Il Matto development board. In common with this device, it has bidirectional data ports, counters and timers, a variety of interrupt sources, and is programmable over ISP.

Re-familiarise yourself with your notes from the first-year lab on interrupts using the Il Matto. Check the datasheet for the ATmega162 microcontroller.



- ◇ ? What types of interrupt are facilitated by the megaAVR microcontrollers?   
How many interrupts does the ATmega162 have? 
- ◇ ? What is "GICR"? 

Work through the code for `inttest.c`, and make sure you understand how it works.

- ◇ ? What are the interrupt vectors for the 2nd external interrupt, the pin change interrupt on PORT A, and the timer overflow from timer 1? 
- ◇ ? Looking at the code for `inttest.c`, sketch the waveforms that you expect to occur on `PORTB[0:2]` just before and after an interrupt. Assume that the time between interrupts is long compared with the time to service the interrupt. 





## 2.2 Super-loops

Consider a simple case where you want to light LED A every 250ms for 125ms, and LED B every ~333ms for 100ms. This should be contained within a super-loop (i.e. a single infinitely-running loop).

- ❖ Write pseudo-code for this. Assume that a `delay` function is available. 
- ❖ Modify the program to light LED A every 500ms for 25ms, and LED B every 100ms for 20ms. Comment on how easy it is to modify the original program. 


## 2.3 Real-time Operating Systems

In this lab you will be using FreeRTOS, an open-source RTOS. Documentation can be found at <http://www.freertos.org/>. FreeRTOS is a simple portable operating system (OS) and is suitable for larger AVRs, PICs and ARM processors among others. At least 20 different architectures are supported. FreeRTOS is easily customised to provide the minimum functionality required for any application. Its only real requirement is the facility to generate a steady set of interrupts or ticks as a real time clock.

- ❖ In the context of operating systems and microcontrollers, define the following terms: *program, task, thread, context switch, pre-emptive multi-tasking, co-operative multi-tasking, blocking, priority, scheduler, timers, interrupts*. 
-  Open the `basic.c` program from the lab web page. Explain how it works. Note down the FreeRTOS function calls. Print the program out, stick it in your logbook, and annotate it.
-  Open the `int.c` program from the lab web page. Explain how it differs from `basic.c` and how this is important.
- ❖ Why are interrupts important for the operation of real-time operating systems? 

## 2.4 Controlling the Operation of a Lift

You will be controlling the operation of a miniature lift. A schematic (`liftpass_sch.pdf`) of the lift connection module is available. Example code, based on FreeRTOS, is available from the labs website (named `lift.c`, `lift.h`, `lift_code.c`, `util.c`). This is for a lift that can go between two levels. Make sure that you understand how the code works.

-  Work through the state machine, and adapt the code, so that the lift can go between three floors.

---

## 3 Laboratory Work

### 3.1 Interrupts and timers

The WinAVR package provides good support for interrupts. The code in Listing 1 shows how. The compiler provides the macro command `ISR(<interrupt type>)` and a range of constants which together generate the ISR and physically link it to the correct interrupt vector. This is done in such a way as to be device independent. Differences in the layout of the interrupt vector table are handled transparently by the compiler.

```
1 # include <avr/ interrupt .h>
2 volatile unsigned char count =0; // stop compiler optimization
3 ISR ( INT0_vect ){
4 count ++;
5 }
```

### Listing 1 An example of an interrupt service routine

Download the files named `inttest.c`, `Makefile`, and `ecsmacros.h` from the M2 web page, and extract them to a working directory.

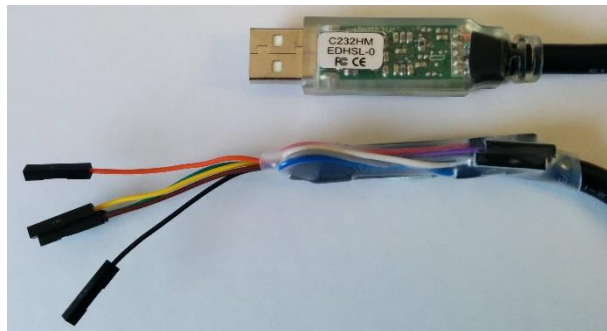
You will use WinAVR to compile code for and program the ATmega644P microcontroller; the makefile is provided for this purpose. To program the AVR, you need to open a DOS Shell in your working directory. Right-click on the directory name in Explorer. One of the options is called "DOS Shells", click on this and you will be given a list of options. Click on the "AVR" option (or similar). A window will then open, set up for the correct version of WinAVR.

```
1 # Target file name (without extension)
2 #~ TARGET = int1
3 TARGET = inttest
4 #~ TARGET = timer
```

### Listing 2 Edit this part of the makefile to build particular programs











Apart from uncommenting the program you want to build, you do not need to make any other changes to the makefile. Typing `make` runs the compiler and generates a `.hex` file for the program. Type `make clean` to delete temporary files before you move to a new program.

You should use the smart serial cable, based around the C232HM chip from FTDI, to program standalone AVR microcontrollers. Only the required connections are exposed (Figure 1). You can find details about how to use this attached to the first-year experiment X2 <https://secure.ecs.soton.ac.uk/notes/ellabs/1/x2/>. You will need to consult the notes on "[Installation details for the FTDI driver](#)" and follow these instructions to set up the programmer. You should also look at Appendices D and F in the [X2 Lab Notes](#).



**Figure 1 FTDI C232HM Cable**

The AVR microcontroller is programmable over an ISP interface. Table 1 shows connections from the C232HM cable. Use short lengths of wire to connect the C232HM cable to the breadboard. You should consult the microcontroller datasheet to find out which pins these signals should connect to. Vcc is not available from the cable, you need to use the Digital Test Bed for this. Ensure that ground is connected correctly.

Pin	Colour		JTAG	SPI	I <sup>2</sup> C	ISP	UART
1	Red		VCC	VCC	VCC	VCC	VCC
2	Orange		TCK	SK	SCL	SCK	TXD
3	Yellow		TDI	DO	SDA	MOSI	RXD
4	Green		TDO	DI	SDA	MISO	$\overline{\text{RTS}}$
5	Brown		TMS	CS	—	RST	$\overline{\text{CTS}}$
6	Gray		GPIOL0	GPIOL0	GPIOL0	GPIOL0	$\overline{\text{DTR}}$
7	Purple		GPIOL1	GPIOL1	GPIOL1	GPIOL1	$\overline{\text{DSR}}$
8	White		GPIOL2	GPIOL2	GPIOL2	GPIOL2	$\overline{\text{DCD}}$
9	Blue		GPIOL3	GPIOL3	GPIOL3	GPIOL3	$\overline{\text{RI}}$
10	Black		GND	GND	GND	GND	GND

**Table 1 C232HM cable connections**

The following command line program will program the ATmega162 with the file `inttest.hex` (which was generated from `inttest.c`):

```
avrdude -c c232hm -p m162 -U flash:w:inttest.hex
```

Note that these commands are case sensitive. In M2, there is no requirement to set the fuses.



Verify the behaviour of the `inttest` program. Observe and sketch (or grab a screenshot of) its behaviour on the scope and comment on whether it agrees with the waveform you sketched in the preparation.

### 3.2 Real-Time Operating System

Aside from microcontrollers, we virtually never write or use code that is tied directly to processor hardware. An OS normally sits between our executing programs and the hardware. Not only do we have an OS (e.g. Windows 7, Mac OSX, or Linux) we also have some form of user interface allowing us to control the computer (Windows 7, Mac OSX or KDE) or a command shell (cmd shell or bash shell). These allow us to: launch and terminate programs; interact with the programs; respond to hardware requirements and to schedule things to happen at different times. An example of the latter would be a virus checker on your PC scheduled to run at midnight. We also have the concept of a background task, which might be a backup task executing in idle moments.

The drawback of these systems is the amount of code required, at least 100's of megabytes. This is certainly not suitable for executing on a resource-constrained system like an AVR microcontroller. However, unless the application is really trivial, we need to organise tasks, respond to events and make things happen at specific times.

So far in this exercise, the programs could equally well be implemented on an FPGA. These programs have generally been small and very responsive. A simple two-floor lift controller could be implemented in as little as 230 bytes of program memory. This is almost a waste given that the Atmega162 has 16K bytes of program memory!

In exploiting the available memory we will probably write programs that are more complex, possibly less responsive and are certainly harder to debug. Consider the 'super loop' problem you looked at in the preparation, where you had to light LEDs at different intervals, and then adjust the intervals. For each transition of pins, a different delay had to be calculated. However, if we used an OS then we could control each LED with an individual task (e.g. Listing 3).

```
1  main
2      start task1
3      start task 2
4      do tasks forever
5  task 1
6      loop
7          set pin 0
8          delay 250
9          clear pin 0
10         delay 250
11 task 2
12     loop
13         set pin 1
14         delay 333
15         clear pin 1
16         delay 333
```

### Listing 3 Independently-timed events using an operating system

Given that task 1 and task 2 can execute independently, then once they are started the operating system takes care of waking them up to turn the LEDs off and on. The operating system is providing a layer of abstraction above the hardware. We can write tasks that appear to be totally independent and the operating system services take care of all the messy things like working out when the task should run next.

You can understand the pseudo code in Listing 3 by interpreting the delay instruction as a command to the operating system to put the current task to sleep and wake it up after the desired number of ticks. Conceptually, this is very simple: the operating system gets a tick at a regular rate from a timer interrupt. It uses this to maintain a real time clock, in units of ticks. Since ticks are interrupts, the OS is guaranteed to be in charge. It maintains a list of tasks, all but one of which are sleeping. It knows which task is currently running, so it is able to save the state of the task (mostly register values) and put this in the list of sleeping tasks, with a message to wake up as soon as possible. Now it runs the next task. This decision is normally made on the basis of priority, and for tasks of equal priority the OS tries to be fair and runs the one that has been waiting the longest.

It is easy to check to see if a task is scheduled to wake up on that tick, and if so let it run for at least 1 tick. Of course, this is significantly more complex than I have described, but think of the OS in terms of a policeman directing traffic at a complicated junction where only one car can go at a time. It may happen very quickly and with complex rules, but this is not a bad model.

Download and unpack the `rtos.zip` file from the experiment web page. Do not edit the make files. Check that the directory structure is correctly unpacked as shown below.


```
rtos
  basic # Simple flashing lights
  interrupts # Shows how to use interrupts properly
  lift # A simple 2 floor lift program
  rtos # The code for the OS is here, don't change anything
```

**Basic** This is the simplest and has two LEDs flashing at different rates.


**Lift** This shows how to write a lift control task that polls the lift signals periodically and uses the FSM to generate suitable controls.



**Interrupt or interrupt** Shows how to pass messages from an interrupt routine to another task, while still flashing LEDs.

-  Work through the `basic` and `interrupts` programs, and determine what the FreeRTOS function calls do.

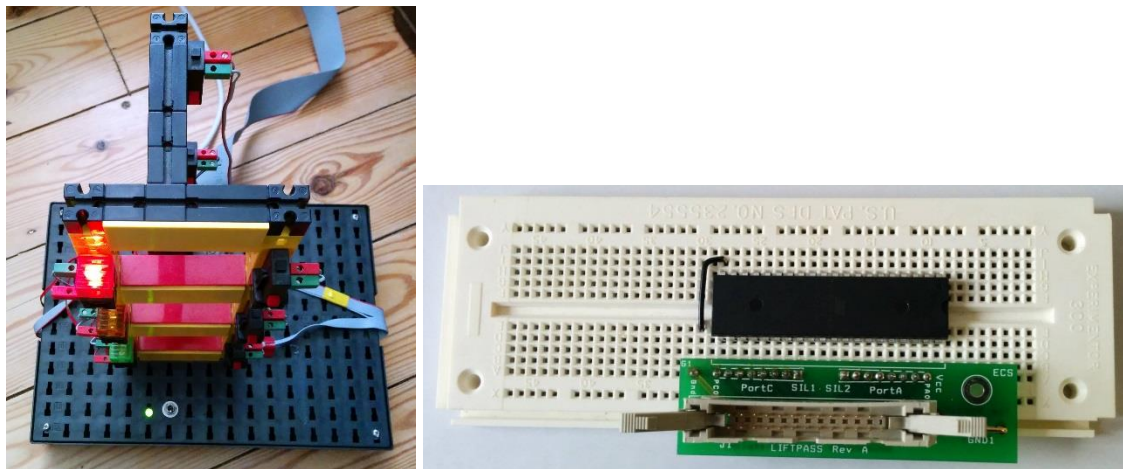
Compile and run only the `basic` and `interrupts` programs, and confirm that they work as you understand. Connect the outputs from `basic` to an oscilloscope (and optionally to LEDs). For `interrupts`, connect outputs to oscilloscope inputs (or optionally to LEDs) and connect the interrupt 0 pin to a switch on the Digital Test Bed.

-  Observe and sketch the behaviour of these programs.

### 3.3 Controlling a Lift with a RTOS

Note that this uses three files to contain the code. One (`lift.c`) is the top level and sets everything in motion. Another (`util.c`) contains code useful in many places and is the start of a utility library that could be expanded for any complex task. Finally (`lift_code.c`) is a rudimentary controller. It reads the status of the lift every 20th of a second and changes the control signals obeying a simple state machine. Note that it starts in a state corresponding to going down to floor zero, so the lift may momentarily attempt to drive down beyond the bottom floor on start-up.

The ECS lift connection module (Figure 2) maps the ATmega162's PORTA and PORTC to the lift cable. There may be other modules inserted into your breadboard; you do not need these for this exercise. Ensure that the connection module is located correctly, make any additional power connections that are required, and connect the lift to the Atmega162 with the adapter provided.




**Figure 2 (a) lift, and (b) ATmega162 and Lift Connection Module**

Compile and run the `lift` program. Ensure that it operates correctly.

-  Observe and note the functionality of the basic `lift` program.

In the preparation, you modified the two-floor program to operate across three floors. Compile and run this program, and debug as necessary.

-  Observe and note the functionality of your modified three-floor program.

Next, instead of periodically polling the inputs every 20th of a second, adapt the program to make the lift controller respond to an *interrupt* if any of the inputs change.





Note down how you intend to modify the program. Make the modifications, and compile and debug the code. Record your test results.

---

#### 4 Optional Additional Work

Marks will only be awarded for this section if you have already completed all of Section 3 to an excellent standard and with excellent understanding.

You have three options for this part:

1. Add some of these improvements: add a queue of button presses; add an external interrupt that resets the lift and returns it to the ground floor. Note that by checking the queue's contents, rather than blocking on them, you can direct results from more than one interrupt to the same task.
2. Explore some of the more advanced operations provided by FreeRTOS. Incorporate these into the lift control program to improve its performance or add more features.
3. Adapt your program to control the operation of two lifts. You should team up with another pair, or obtain a spare lift (if there is one) to test your code.



Remember to fully record your design steps and the results you have obtained.

---