Low Power and Embedded Systems - Workbook 1

Table of Contents

Introduction 1
- Document conventions 2
- Supporting material 2
- Workbooks structure 3
- What is Needed 3

Exercises 4
- Exercise 1 5
- Exercise 2 - Flash an LED 8

Introduction

The aim of this module is to teach students how to design and write programs for low power embedded devices that communicate with other computers or devices to solve practical problems. The course will cover the relevant theoretical knowledge required for writing such applications, and provide practical experience writing C programs for the Atmel family of microcontrollers. The module will also give students an appreciation for some of the research issues in low power enbedded systems.

For the first 5 weeks of this 8 week course, you will be provided with a workbook like this one. It contains step by step instructions for you to follow, and links to reference material. Each workbook contains a number of exercises. Completion of these exercises must be verified by a demonstrator who will award a 'tick' for each completed workbook. In some instances you may need to provide completed C code. When this is the case, you are required to provide neatly formatted and commented code, such that someone else could take your code and modify it without having to decipher cryptic comments or obscure syntax. For everyone's sanity, use the simplest and most readable way of carrying out the task - there are no bonus points for clever but unreadable code. There may be occasions (eg for optimised speed) where you need to use cunning optimisations and tricks. If so comment them really well, or the next programmer to see the code may well decide to make it more readable and break it. That next programmer may well be you in 3 years time, of course.

For the final 3 weeks, you will be conducting your own design and build project. There is a list of project suggestions available: project_suggestions.html [http://www.cl.cam.ac.uk/teaching/0910/P31/project_suggestions.html] or project_suggestions.pdf If you have any ideas which you think might be suitable for a project, please discuss them with us at the earliest opportunity, so that we can judge their feasibility, make recommendations and order any specialist components. You need to have a 1-page project proposal approved by a demonstrator prior to session 6.

Work your way through the exercises in each workbook. Once you have done this, contact a demonstrator for 'ticking'. You are expected to have completed each workbook before the start of the next session. During the sessions, demonstrators are available to assist you if you have any questions, or if anything is not clear.

An on-line version of this guide is available at:

workbook1.html [http://www.cl.cam.ac.uk/teaching/0910/P31/workbook1.html]

You should check this page regularly for announcements and errata. You might find it useful to refer to the on-line version of this guide in order to follow any provided web links or to cut and paste example code.

Assessment

You need to have obtained 5 'ticks' awarded for completion of each of the 5 workbooks. You will need to submit a formal report concerning your project work. The report should be approximately 4,000 words in length, and should be submitted to the Teaching Office on the first teaching day of the Lent Term. Students are also encouraged to submit their lab books.

Document conventions

Within these worksheets, the following conventions will be used.

Important This style is used for information of particular importance.

```
This style is used for program listings.
Note that you can save yourself considerable amounts of typing by
referring to the online version of this document and using cut and paste.
```

This style is used for instructions for you to follow.

When referring to the names of registers in the microcontroller or bit positions within registers a `fixed spaced` font will be used.

This workbook was created using Docbook, and transformed into .pdf and html versions.

Supporting material

All the following are available at http://www.cl.cam.ac.uk/teaching/0910/P31/ or subdirectories.

The online version of this workbook at http://www.cl.cam.ac.uk/teaching/0910/P31/workbook1.html allows you to follow the links easily.

template.c	http://www.cl.cam.ac.uk/teaching/0910/P31/code/template.c A template suggestion for C programs. You may already have your own style for C programs, in which case that would be a good choice. The important things is to develop a consistent structure. When designing with microcontrollers you will find that you re-use existing code a great deal. Having a structure to your programs will help to make code re-use easier, and less prone to errors, for example forgetting to include initialisation code.
atmega168P.pdf	http://www.cl.cam.ac.uk/teaching/0910/P31/docs/atmega168P.pdf Data sheet for the Atmel ATMEGA168P used in these exercises. You will need to refer to this frequently. Within these workbooks it will often be referred to as 'the datasheet' The section numbers referred to in these workbooks refer to revision 8161C of the datasheet dated 05/09.
C Types	http://www.cl.cam.ac.uk/teaching/0910/P31/docs/c_types.txt An overview of the C types used in gcc-avr.
Makefile	http://www.cl.cam.ac.uk/teaching/0910/P31/code/workbook1_Makefile

	A Makefile template used in Exercise 1.
avr-libc documentation	http://www.nongnu.org/avr-libc/
	Documentation for libraries available on the PCs for the Atmel range of microcontrollers.
breadboard_program.pdf	http://www.cl.cam.ac.uk/teaching/0910/P31/docs/ breadboard_program.pdf
	A circuit diagram for connecting a programming header to the microcontroller.
Programming header	http://www.cl.cam.ac.uk/teaching/0910/P31/docs/ programming_header.jpg
	Photograph showing the connections for in-circuit programming of the microcontroller.
AVR-GCC tutorial	http://winavr.scienceprog.com/avr-gcc-tutorial/
	Tutorial for AVR-GCC.

Workbooks structure

Workbook 1 will cover Microcontroller Clock sources, and very basic I/O.

Workbook 2 will cover basic Serial communication, Serial communication using interrupts, and the use of the Analogue to Digital converter.

Workbook 3 will cover the use of timers in the microcontroller, and adding an LCD display for more readable output.

Workbook 4 will cover interrupts generated from transitions on inputs, and use these to create a precision interval display. It will also cover the sleep modes for the microcontroller.

Workbook 5 will cover the use of libraries, the SPI interface, and adding bulk memory by building up from low level I/O functions.

What is Needed

Hardware

- A microcontroller Integrated Circuit (IC) - In these workbooks you will be using the ATMEGA168P microcontroller made by Atmel. There are a large number of members of this family of microcontrollers each with different capabilities. For the project part of the course, other devices in the family may be a better fit to the requirements.
- A power supply (PSU). The early exercises need a PSU capable of delivering 5 V at 100mA, later ones will use 3.3 V. Note that some members of the microcontroller family will work at reduced clock speeds at supplies down to 1.8 V.
- A programmer for the microcontroller - these worksheets are written assuming the use of a USB based device from tuxgraphics.org
- A prototyping board so we can develop the hardware, sensors and interface circuits.

Software Toolchain

- avr-gcc - a C cross compiler.

An alternative would be to use an AVR cross assembler, enabling us to write assembly code directly. While an assembler might produce slightly faster and more compact code, C should give reasonable results whilst being somewhat easier to program. We will use the gcc cross-compiler. The cross prefix refers to the fact that we run it on one type of computer (Intel x86 architecture) but create code to run on a different architecture (the Atmel device).

- avr-libc - libraries for the Atmel AVR family of microcontrollers we are using. Documentation is available at http://www.nongnu.org/avr-libc/.
- binutils-avr - compiler tools such as 'make'.
- avrdude - software to control the programmer.

Number notations

- Decimal numbers have no prefix and are shown as e.g., 45
- Hexadecimal values are shown in the form 0xDF
- In these worksheets we will not use octal. It is perfectly valid, but is likely to cause errors.

C Compiler Workflow

In general, each of the following steps need to be performed to create and run software on the microcontroller.

- The C code is created in a file ending with the extension .c
- A program called `make` is used together with a configuration `Makefile`. `make` uses the timestamp on files and the contents of the `Makefile` to decide which commands need to be run. `make` knows nothing about the compile process - it simply compares timestamps on input and output files, and runs a command if the input is newer than the output. The resulting output file may be an input to another line in the `Makefile`, and so the process may repeat multiple times. If there are no errors, then eventually all output files will be up to date. The input files are more generally described as `source` files and the output files are described as `object` files.

  During the procesing by make, the .c files (there may be several) will be compiled, and `linked` with library object files to form `object` files, ending in .o

  These .o files are converted in turn by `make` to a suitable format for the programming hardware, usually with a .hex extension.
- The command `make program` invokes a program called avrdude which transfers the .hex file to the microcontroller.

  If the code transfers correctly, the microcontroller restarts and executes the code immediately, exactly as it would from power on.

Exercises

This week there are two exercises. They are intended to be straightforward, but they will provide a thorough test of the toolchain being used, and give you a first taste of programming these microcontrollers in C. These examples are also useful later on since they will enable you to verify the hardware and software toolchain when more complex programs that you have developed fail to work as expected.

`Exercise 1`	Set up the clock to the microcontroller, and verify its function using an oscilloscope.
`Exercise 2`	Flash a light emitting diode (LED) once per second.

Exercise 1

Setting-up the clock for the microcontroller.

The ATMEGA168P device can use one of several clock sources, including one generated internally. The internal clock isn't very precise, but for many applications it is sufficient. An option exists to output the clock to a pin, where its function can be verified using an oscilloscope.

In this exercise you will determine suitable values for `hfuse` and `lfuse` such that the devices use the clock generated within the device, and also output it to a pin where you will be able to verify that it is working as expected. Note that the term 'fuse' is left over from earlier programmable devices which really did have fuses which, once blown, were fixed in that state. In newer devices, the term refers to settings made in flash memory which can be reprogrammed multiple times by use of the programming hardware, but importantly, cannot be changed by the program running on the microntroller itself. The factory default for hfuse is correct for what we require here, but you will need to make one change to lfuse.

For this exercise, refer to the datasheet for the device at http://www.cl.cam.ac.uk/teaching/0910/P31/docs/atmega168P.pdf, in particular sections 1 (pin configurations), 6.2 (System clock and clock options), and 25.2 (Fuse bits) and work out the hexadecimal values for lfuse and hfuse values to make the device

- use the Calibrated Internal RC Oscillator.
- divide the clock by 8.
- output the clock to a pin.

CHECK the values for lfuse and hfuse with a demonstrator. They will be used later.

Creating a directory structure to hold your work

Many of the exercises in these worksheets will build on example code and on code created in previous exercises. Creating a clear directory structure will help both you and the demonstrators. For each exercise, make a COPY of relevant files from previous exercises.

In some instances, example code or partially completed code will be provided for you.

Log in to the computer, and create a directory structure. Commands are listed immediately below. Use cut and paste if you can.

```
cd ~
mkdir -p embedded_systems/workbook1/exercise1
mkdir -p embedded_systems/workbook1/exercise2
mkdir -p embedded_systems/workbook2/exercise1
mkdir -p embedded_systems/workbook2/exercise2
mkdir -p embedded_systems/workbook2/exercise3
mkdir -p embedded_systems/workbook3/exercise1
mkdir -p embedded_systems/workbook3/exercise2
mkdir -p embedded_systems/workbook4/exercise1
mkdir -p embedded_systems/workbook4/exercise2
mkdir -p embedded_systems/workbook5/exercise1
mkdir -p embedded_systems/workbook5/exercise2
mkdir -p embedded_systems/workbook5/exercise3
cd embedded_systems/workbook1/exercise1
```

The Programmer

Plug the programmer in to your PC, and wait a few seconds for the PC to recognise that a new device has been connected.

Type `dmesg`. The last line should read something like the following:

```
[2323862.498398] usb 5-1: FTDI USB Serial Device converter now attached
to ttyUSB0
```

The key item here is the reference to `ttyUSB0`. The programmer is now available on `/dev/ttyUSB0`. Depending on the USB devices connected to the PC, the Serial converter may be connected to /dev/ttyUSB1 or a higher number.

The Makefile

cd to ~/embedded_systems/workboook1/exercise1

Create a file called `Makefile` using your favourite editor to read as below. Note that Emacs 22 and Gedit (shown as 'Text Editor') are available from the desktop, Pico is available from a terminal window.

The name must be exactly as shown, i.e., Upper case M, the rest in lower case.

Use cut and paste from the online version of this workbook at http://www.cl.cam.ac.uk/teaching/0910/P31/workbook1.html, or right click and choose 'Save as' from http://www.cl.cam.ac.uk/teaching/0910/P31/code/workbook1_Makefile.

Note: Makefiles use tabs not spaces for the whitespace preceding commands. In the listing these are shown as `/t`.

The contents of the Makefile will be explained in more detail later on. For this exercise only the last two lines matter, but exercise 2 will use the rest of the Makefile.

```
CPPFLAGS=-I. -I../lib
MCU=atmega168
VPATH=../lib
all: exercise1.hex

exercise1.elf: exercise1.o

%.o: %.c
 avr-gcc ${CPPFLAGS} -Os -mmcu=${MCU} -o $@ -c $^

%.elf: %.o
 avr-gcc -Os -mmcu=${MCU} -o $@ $^

%.hex: %.elf
 avr-objcopy -j .text -j .data -O ihex $^ $@

%.lst: %.elf
 avr-objdump -h -S $^ > $@

clean:
 rm -f *.o *.elf *.hex *.lst


program: exercise1.hex
 avrdude -p m168p -P /dev/ttyUSB0 -c avrusb500 -e -U flash:w:$^

fuses:
 avrdude -p m168p -P /dev/ttyUSB0 -c avrusb500 -e -U hfuse:w:0xDF:m
 avrdude -p m168p -P /dev/ttyUSB0 -c avrusb500 -e -U lfuse:w:0x62:m
```

The Makefile you have just created contains the default settings for the hfuse 0xDF and lfuse 0x62.

Edit the hfuse and lfuse values to those you worked out and checked earlier with a demonstrator.

You might also need to edit the references to /dev/ttyUSB0 to match the those of actual device. See the previous section 'The Programmer'.

Here is a brief explanation of the line you have just edited (note: this is only for information).

`avrdude`	The linux program used to program the device.
`-p m168p`	Tells avrdude the device type being programmed - the device signature is checked before programming.
`-P /dev/ttyUSB0`	Tells avrdude where the programmer is attached.
`-c avrusb500`	Tells avrdude the type of programmer being used.
`-e`	Erase the device before programming.
`-U hfuse:w:0xDF:m`	Perform a memory operation.
	In this case hfuse:write:value-to-write:immediate (that is the value to write is literally the value 0xDF).

Connections

The next stage is to fit the microcontroller into the prototyping board, and wire it up to a PSU and a programmer. These devices can use In Circuit Programming - i.e., the device does not need to be removed from the board in order to be programmed. Note that Pin 1 of the device is marked with a dot, and the pins are numbered counter-clockwise looking from above.

Refer to the circuit diagram http://www.cl.cam.ac.uk/teaching/0910/P31/docs/breadboard_program.pdf. Connect the device up as shown in the diagram. There is a photo showing one possible layout here http://www.cl.cam.ac.uk/teaching/0910/P31/docs/programming_header.jpg

Important The 10uF tantalum capacitor is polarised. It MUST be fitted the right way round. Tantalum capacitors rely on a thin electrolytic film to achieve their high capacitance per unit volume, and this film depends on the electric field generated by the applied voltage. Connecting an electrolytic capacitor incorrectly can lead to the device exploding.

Apply 5 V from the PSU to the device.

the command to program the device fuses is:

```
make fuses
```

The programming software carries out a read (to get the device type), a write (to write the fuses) then a read (to verify the fuses) for each of the fuses.

If all went well, then you should be able to verify using an oscilloscope that there is a 1MHz square wave on PORT `PB0`, i.e., pin 14 of the device.

If not, the following list gives some common problems

1. No power to the microcontroller.
2. The microcontroller is in the board the wrong way round.
3. Not looking at the correct output pin.
4. The oscilloscope probe ground clip is not connected to the board ground (0V).
5. Wrong values for hfuse and lfuse - if you get the value wrong and then can't program the device, it may be that you have set it to use an external clock, or have disabled programming. Either way you'll need a new ATMEGA168.

Exercise 2 - Flash an LED

So far you may not feel that much progress has been made, but now we know that:

1. The Atmel device is receiving power and using an internal clock.
2. The in-circuit programming is working.
3. The software toolchain and at least the last 2 lines of the Makefile are correct.

In this exercise, you will attach an LED to an output, and write some C code to flash the LED.

To simplify this task, a template is provided for you, but some terms need to be defined first.

Input / Output Notation

For a detailed description, refer to section 11 of the datasheet for the device. The following is a brief summary:

Each 8 bit IO port is controlled by three 8 bit registers: DDR (Data Direction Register), PORT (data output register), PIN (data input register)

Different devices have different numbers of ports referred to as `PORTA, PORTB, PORTC` etc, and controlled by `DDRA, DDRB, DDRC` etc

DDR

The 8 bit Data Direction Register, `DDR`, contains a 1 in each position where an output is required, for example to set the top bit (bit 7) as an output and the rest as inputs:

```
DDRB = 0x80 ;
```

PORT

Assigning an 8 bit value to a PORT will write a value to the Data Output register. If the corresponding pin is set to be an output, then the value in the Data Output register will control the output pin.

To change fewer than 8 bits at once, use the bitwise AND and bitwise OR functions in C (see later in this section).

For example to set the top 5 bits to a low state, the lower 3 bits to a high state

```
PORTB = 0x07 ;
```

PIN

PIN is used when referring to the data input register. It shows the value from the pin on the device whether or not that pin is defined as an input or an output. For example, to read the values on PORTB:

```
input_variable = PINB ;
```

Hardware

For this exercise, LEDs with suitable resistors built in for 5V operation are provided for you (the green and red ones).

For these devices, the cathode, is the shorter lead. Connect it to Ground.

Connect the longer lead on the LED to `PB1` (pin 15 on the ATMEGA168).

Software

In general it is a good idea to have structure in your programs. For microcontrollers, it makes a lot of sense to have building blocks which you can cut and paste between your different programs. Microcontrollers don't have a vast amount of code space so you can't simply include all possible functions just in case you need them. A suitable template is provided: `template.c [http://www.cl.cam.ac.uk/teaching/0910/P31/code/template.c]`

We have just seen the method used to assign ports as inputs and outputs, and how to read and write to them.

When using microcontrollers it is often a requirement to set (or clear) one bit of a port at a time, rather than writing a new value to the whole port. The preferred way to set a bit is to use the |= operator. For example bit 5 of PORTB:

```
PORTB |= (1<<PB5);
```

and to clear that bit

```
PORTB &= ~(1<<PB5);
```

To explain this in more detail: In the library header <avr/io.h> PB5 is defined as the value 5. So (1<<PB5) is therefore 0x20, or 00100000 in binary

```
PORTB |= 0x20
```

is equivalent to

```
PORTB = PORTB | 0x20
```

and will set bit 5, leaving all others untouched. Remember that the bits are labelled from bit 7 (on the left) to bit 0 (on the right).

We actually only need the brackets round (1<<PB5) in the case ~(1<<PB5) in order to define the operator precedence, but it is good practice to put the brackets in anyway as it is easy to miss when editing code.

Make a copy of the template from `template.c` `[http://www.cl.cam.ac.uk/teaching/0910/P31/code/template.c]` and put it in the directory structure you created earlier. Use the linux `mv` command to rename it to exercise2.c

In the header of your program, add the following

```
#define F_CPU 1E6       // 1MHz
#include <util/delay_basic.h>
#include <util/delay.h>
```

You now have access to the library functions `_delay_ms(J)` and `_delay_us(K)` where you subsitute (16 bit) integer values for J and K, which must be known at compile time. You can't use variables for J and K; if you try you will just get a very small or a very large delay.

One last step.

In order to loop forever (rare in most C programs, but common when using microcontrollers)

```
while (1) {
// code in here
}
```

you now have all the parts you need to write code to flash the LED once per second.

Modify the `port_direction_init()` function to assign PB1 as an output

Write a chunk of C code in the main() function to flash an LED once per second. In pseudo code it would be:

```
    while {
       Set a suitable bit in the PORT register to output a 1 to the LED
       wait(0.5 second)
       clear a suitable bit in the PORT register to output a 0 to the LED
       wait(0.5 second)
      }
```

assuming you called this exercise2.c, modify the Makefile to change exercise1 to exercise2 (4 places)

Connect the programming header and issue the following commands:

```
 make



 make program
```

You should be rewarded with a flashing LED. If it isn't flashing but you think your code is correct, you can look at pin 15 with an oscilloscope or multimeter to check.