The AVR Microcontroller

Introduction

Digital electronic circuits, which process information represented as numbers rather than as voltages, can be found not only in computers but also in just about every kind of electrical device. In many cases, the digital circuits used are not custom-made for each application, but are general-purpose computers that have been programmed to perform the desired function. The AVR microcontroller used in this experiment is one such device. In this experiment you will program the AVR using the C language, to make timed pulses, and control some displays. You should come to appreciate how a microcontroller can be useful in experimental work (e.g. year 2 projects).

1 A quick description of the AVR[1, 2]

1.1 Overview

The ATMEGA32U2 is one of a broad family of similar chips made by Atmel corp. They can be considered as "computers on a chip", since they have memory, a central processor and input and output devices. In this experiment, we are using a Minimus module, which holds the chip and provides easy connections to it. Table 1 compares the Minimus with the lab PCs.

Parameter	Lab PC	Minimus	units
Processor speed	$\sim 10^9$	$\sim 10^6$	instructions/sec
Permanent memory	$\sim 10^{11}$(disk)	$\sim 32 \times 10^3$	bytes
Working memory	$\sim 10^9$(RAM)	$\sim 10^3$	bytes
Interfaces	Many	Logic signals, USB	
Cost	400	5	£
Weight	10	0.007	kg
Power consumption	200	10^{-6} to 1	watts

Table 1: Comparison of the AVR Minimus module with a PC

1.2 Programming

The AVR can be programmed in C. However, the programmer should bear in mind not only the small memory of the chip, but also that its fundamental data type is the 8-bit byte (in C, a **`char`**), and that using an **`int`** or a **`double`** (32 or 64 bit variables) is more expensive in both memory space and speed, as is arithmetic other than integer addition and subtraction. With thoughtful programming, these are not significant limitations for most applications in the lab. For extreme cases, one can get much more capable chips (and much less capable ones, if manufacturing cost really matters).

[1] "Alf (Bogen) and Vegard (Wollan)'s Risc processor" http://youtu.be/HrydNwAxbcY

[2] The manufacturer's data sheet runs to hundreds of pages, and one has to study the relevant part in order to use an unfamilar facility http://www.atmel.com/devices/atmega32u2.aspx

1.3 Peripherals

The AVR has several internal circuits for communications, timing, measurement, and so on. It can be configured to connect the required ones to its limited number of external pins. Here are the peripherals that we will use in this experiment.

1.3.1 Input/Output (I/O) ports

The number stored in some predefined **`char`** variables: **`PORTB`**, **`PORTC`**, and **`PORTD`** can be reproduced in binary as logic level signals on the pins (an output port). A logic 1 is nominally +5 volts; 0 is nominally zero volts. Alternatively, the logic levels applied to the pins can be read as the value of the variable (an input port). Rapid manipulation of these variables ("bit banging") is a useful technique for making signals.

1.3.2 Analogue comparator

Some of the pins can be used to monitor analogue voltages anywhere in the 0 to 5V range. The comparator will give a digital 1 or 0 output depending on whether one applied voltage is less than or greater than a second one (or a fixed reference voltage of 1.1 V). We will use this to measure an analogue signal.

2 The count-forever program

2.1 forever.c

This program (**`forever.c`** in the **`AVRcode`** folder on the lab PCs, reproduced overleaf) makes use of an output port, **`PORTD`**, to drive some LEDs. The value saved in the **`PORTD`** variable is increased by 1 again and again. Most of the code is common setup which you need not understand in detail, but you should look at the part involving **`PORTD`** and try to predict what the individual bits of this variable will be doing. A declaration similar to **`char PORTD;`** is hidden inside the header file.

2.2 Programming the chip.....

Load the program into the Geany editor. Compile it using the Build button (there should be no errors). Connect the programmer module to the PC using the USB cable. Put the chip into programming mode as follows: first press and hold its RESET button, then press the HWB button, release the RESET button, the and finally release the HWB button (see Figure 1). This is easy to do by rolling a finger across the two buttons. You have to do it every time you want to reprogram the chip. Use Geany's Execute button to transfer the program to the chip; again there should be no errors. The program should start running on the chip immediately. It takes power from the USB socket; but all the code runs in the AVR chip.

Geany:  Build:  Execute (program): 

```
// Yr 2 forever counter for Minimus (AVR ATMEGA32U2)
// forever.c
#include <avr/io.h>         // device-specific I/O
#include <avr/wdt.h>        // watchdog timer
#include <avr/power.h>      // system clock
int main(void) {
    // Initialise system clock, disable watchdog timer
    clock_prescale_set(clock_div_1);  // full crystal speed (16 MHz)
    MCUSR &= ~(1 << WDRF);
    wdt_disable();

    // Set the data direction registers
    DDRD = 0xFF;  // all 8 bits of port D as outputs

    PORTD = 0;
    while (1) {   // Loop forever (because 1 is always true)
        PORTD++;
    }
}
```

2.3 Measure the outputs

Connect the zero volts ground reference pin (**GND**) of the module to a rail of the breadboard, and use the oscilloscope to look at the signals on each of the pins of **PORTD: PD0** to **PD7** (Figure 1). Sketch some of the waveforms. How does the timing of each signal relate to its neighbour?

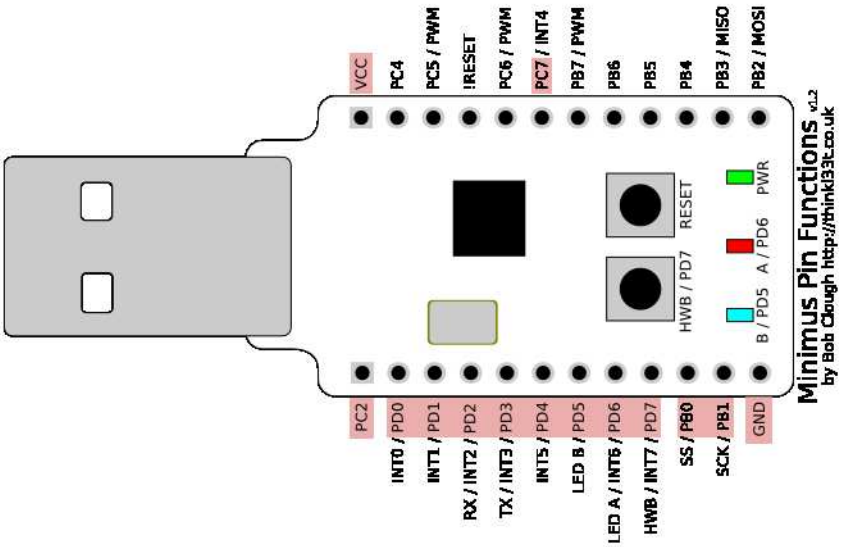

Figure 1: Pinout of the Minimus board. Connections used in this experiment are highlighted.

Write the numbers 0 to 7 in binary in a column:

```
000  (zero)
001  (one)
010  (two)
etc...
```

Q and see if you can find the same pattern. How fast is the fastest signal? What does this tell you about the speed of the program?

Reprogram the chip with **`forever_slow.c`**. This is the almost same program, but every time around the **`while(1)`** loop, a library function **`_delay_us`** is called to waste 10 microseconds. Predict how fast **`PD0`** should be changing, then check with the oscilloscope.

Which of the outputs do you expect to be producing an audible frequency? Check your prediction by connecting a loudspeaker in series with a 47 Ω resistor between **`GND`** and each of **`PD0`** to **`PD7`** in turn.

3 Driving LEDs

Connect the package containing light-emitting diodes (LEDs) to the **`PD0`** to **`PD7`** outputs of the module, making sure there is a 330Ω resistor in series with each LED, as shown in figures 2 and 3. The resistors limit the current in the LEDs to a safe value. For reasons that will become apparent later, place the LEDs mid-way along the breadboard.

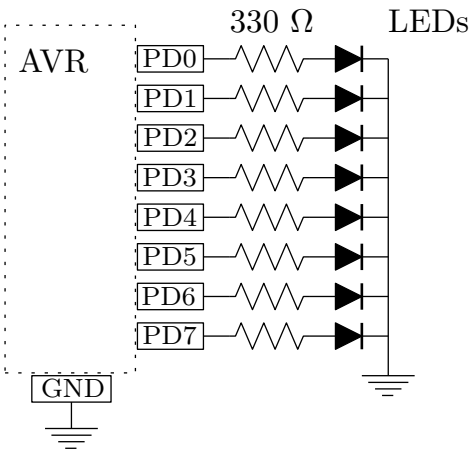

Figure 2: Connection of LEDs

Q Still using the **`forever_slow.c`** program from section 2.3, observe the LEDs. Why are the eight connected LEDs all the same brightness, even though the signals applied to them differ?

In order to see the LEDs flash, the program has to be slowed down some more. Estimate how fast a flashing you will be able to see, then adjust the argument to the **`_delay_us()`** function (a constant number of microsconds) to get the LEDs to flash in that range. (Hint: old TVs update the screen at 50 Hz). Program the chip and see how close you were. You must save "Save As" your modifications with a different name (still ending in **`.c`**) before building, because the original file is write-protected.

Roughly how slowly does an LED have to flash before you can actually see it flash? Use the scope to find out.

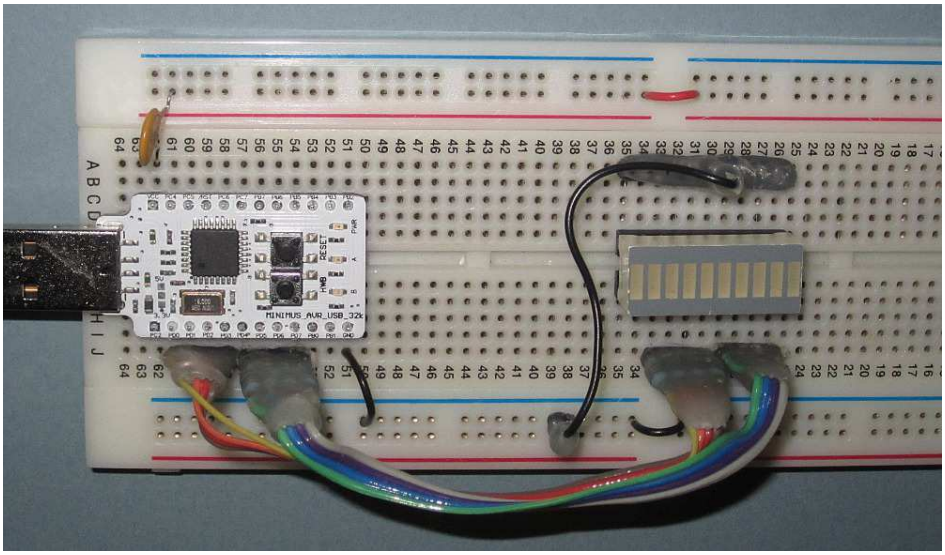

Figure 3: Layout of LED circuit

4 Your name in lights

4.1 Writing letters

By changing the program, the LED circuit can be made to light the LEDs in any sequence you choose. The program **`message.c`** lights the LEDs in a sequence that can appear to write messages in mid-air, exploiting persistence of vision.

Examine **`message.c`** and try to explain how the **`for`** loops in in **`main`** build up a sequence of letters. Program the AVR and swing the breadboard rapidly from side to side, and you should see the letter **`A`** written in the air (hold the board firmly at the AVR end, and shake it in the shadow under the bench for the best chance).

How does the sequence of numbers representing 'A' in the **`letters`** array form the visual impression of the character? You will need to convert the hexadecimal numbers (like 0x1F) to binary. **Q** The table below should help. A pair of hex digits can be read as eight binary digits (0x1F = 0001,1111 binary). The initial 0x just means "this is an 'exadecimal number".

Hexadecimal	Binary	Decimal
0x0	0000	0
0x1	0001	1
0x2	0010	2
0x3	0011	3
0x4	0100	4
0x5	0101	5
0x6	0110	6
0x7	0111	7

Hexadecimal	Binary	Decimal
0x8	1000	8
0x9	1001	9
0xA	1010	10
0xB	1011	11
0xC	1100	12
0xD	1101	13
0xE	1110	14
0xF	1111	15

4.2 The synchronisation problem

Change the message and the corresponding length count in the program to correspond to your name, and try again (remember you have to "Save As"). Short names work best. Unless your name is very symmetric, you will find it harder to read.

The program does not synchronize the LED sequence with the swinging of the breadboard. Consequently, your name gets written both backwards and forwards. Connect an acceleration-sensitive switch to port **`PC7`**, as shown in Figure 4, to help prevent this. The switch should be oriented along the arc of the swing. The rest of the circuit remains the same. Notice that the +5V supply is already connected to the breadboard via a thermal fuse (top left of figure 3). When the switch opens, the bit

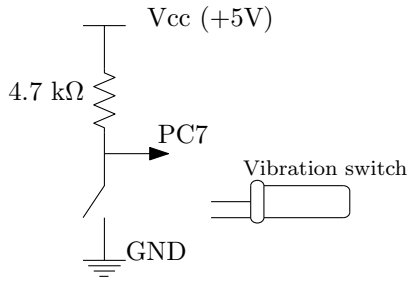

Figure 4: Adding a vibration switch

Q read from **`PC7`** will change from 0 to 1 (why?). A function to detect this, called **`switchsync`**, is included in the program, but you must modify the program to call it, immediately after **`PORTD = 0`** at the top of the main **`while`** loop. Reprogram the chip and try it out. If the message is backwards, reinsert the switch, pointing the other way. You might have to adjust some of the delay times, depending on your name and the style of swing.

Q The **`switchsync`** function has more stages than you might expect. Try to explain why they are necessary.

Figure 5: Persistence of vision display

5 Measuring analogue quantities: light meter

5.1 Timing the charging of a capacitor

In this circuit, a light-dependent resistor (LDR) is used to charge a capacitor, see Figure 6. As the light gets brighter, the resistance of the LDR decreases, so the capacitor charges faster. Inside the AVR, we use a comparator to determine when the capacitor has charged to a fixed reference voltage. We also use a digital output from the AVR (on the same pin) to discharge the capacitor, so the process can be repeated.

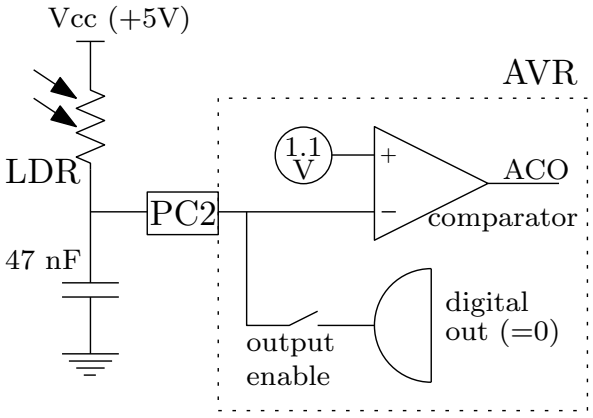

Figure 6: Capacitor charging and discharging circuit

Leaving the wiring for the LED bar in place, add the LDR and capacitor as shown (you can remove the acceleration switch). Program the AVR with **`lightmeter.c`**. Use the oscilloscope to watch the voltage across the capacitor as the circuit operates. You should see a sawtooth waveform, which varies as you shade the LDR. The contents of the LED bar should be a binary representation of the light level. Record your observations and explanations, by referring to the program. The Appendix may help you follow the code. Why is the sawtooth slightly nonlinear? **Q**

5.2 A proper display

A meter really needs a decimal display, not a binary one. Add the four-digit display, as shown in Figure 7. Program the AVR with **`sevenseg.c`** and try out the lightmeter. Only the digits 0 to 4

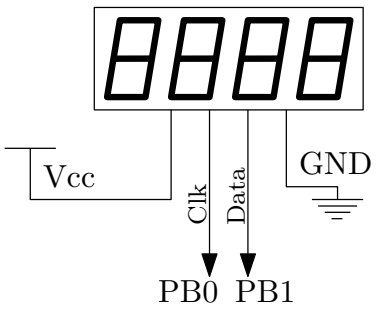

Figure 7: Connection of seven-segment display

will show up, as some work has been left for you to do in the **bcdto7seg** array.

sevenseg.c alters the numerical data twice: first from the single **charge_time** variable to one variable for each of the decimal digits (units, tens, etc.), then each digit is looked up in the **bcdto7seg** array to convert it into the arrangement of bits that lights the correct parts of the seven-segment display, see figure 8. Finally, the bit patterns are bit-banged, one bit at a time, onto the single data wire that goes to the display.

Work out the missing entries in the **bcdto7seg** conversion array, add them to the program and see if your display works.

Q

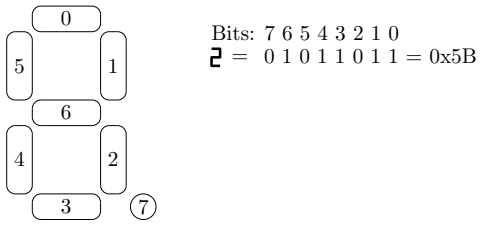

Figure 8: Bit positions in seven-segment display

To show that you know what the different steps do, work out each step in the translation of the binary value 00010101 to the data that goes to the display. How can all the necessary bits be sent along two wires?

Q

5.2.1 Enhancements

If things are going well, try altering the code. For example, you could modify **charge_time** just before it is converted to BCD, so as to get a reading that increases with brightness. You could also change the scale by changing either this, or changing the delay in the charging loop.

You can also try out the program **lightmeter_usb.c**, which sends its data to the PC via the USB. Unfortunately, the code is too long to discuss in this experiment.

6 Checklist

Your lab book should contain:

- Diagrams of circuits you built.
- Programs (or parts of programs) that you wrote or understood, with comments showing how they work — but you don't have to include the full text of programs that are supplied to you.
- Observations and explanations as prompted by this manual.

Appendix

Some help to understand the microcontroller C code

There are quite a lot of comments in the code to explain what it is doing, so try to follow them first. Two detailed features may be unfamiliar to you. First, the names of the registers that are associated

with the hardware features of the AVR. These appear as variables with names like **ACSR**, **DDRD**, and constants like **ACBG**. These are defined in the (300 page!) manual of the AVR chip, and are brought into your code by means of the header file **#include** lines. We have already done the work of finding out exactly which registers and bits to worry about, so you are not expected to. Here are some relevant examples:

DDRD Data direction register for Port D. Every bit set to 1 in here makes the corresponding pin an output (otherwise it is an input). Likewise **DDRB** and **DDRC** for ports B and C.

PORTD The lowest 8 bits of data saved here appear as logic voltages on the Port D pins, if the pins are configured as outputs (using **DDRD**).

ACMUX Analogue comparator multiplexer. The number you store here will determine which input is used as the '-' input of the comparator. 1 means Port C pin 2.

ACSR Analogue comparator status register. Each bit does something different to the comparator. For example, setting bit 6 connects a 1.1 volt bandgap reference to its '+' input, see **ACBG** below.

ACBG Analogue comparator bandgap select. Not a register, just a constant value (6, as it happens). Used to avoid having to remember that bit 6 controls the bandgap reference

ACO Analogue comparator output. This constant is the bit number of the output of the comparator, to save having to remember that it is bit 5.

Bit manipulation in C, or what does `FOO &=~(1<<BAR)` mean

In hardware programming, you often need to alter or read just one bit of a variable (because it corresponds to a logic signal somewhere in the hardware). So for example, to make a number whose binary representation has a single bit set in bit position 4, we use the C expression **1<<4**. This takes the value 1 (**00000001** as a **char** in binary) and shifts it 4 places to the left, resulting in **00010000**. So the expression **foo = 1<<4** puts this sequence of bits in the variable **foo**.

But we often want to set just bit 4 without zeroing all the others. This is done by the expression **foo |= 1<<4**. This bitwise-or operator works because or-ing a value with 0 leaves it unchanged but or-ing with 1 results in a 1.

Now suppose we want to clear (i.e. set to zero) just bit 4, leaving the others alone. We first want a value with ones everywhere *except* at the location we want to clear. The bitwise invert operator, **~** does that: we first form **~(1<<4)**, which is **11101111**. This we bitwise-and into the existing value: **foo &= ~(1<<4)**, which works because and-ing with 1 leaves bits unchanged.

Finally, we define a couple of C macros to make the bulk of our code clearer:

```
#define bit_set(r,b) r |= (1<<(b))
#define bit_clear(r,b) r &= ~(1<<(b))
```

so we can write **bit_clear(PORTC, 2)**, which will be automatically expanded to the expression that clears just bit 2 of the **PORTC** register. Similar macros let us write the boolean expression **bit_is_set(register, bitnumber)** which will evaluate to true or false.

MSC 2013-10-03