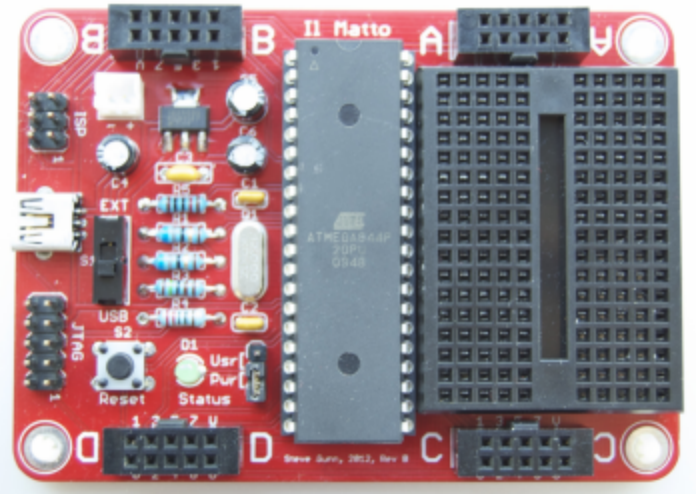
Embedded C Programming Logbook
Joseph Butterworth



Contents

Setting Up the Embedded ToolChain

Download WinAVR and install build/debugging tools

Download AVRDUDE 6.3 programming tool (new link with additional files)

Extract all the files from the zip into the WinAVR bin directory (**`C:\WinAVR - 20100110\bin`**). You will need to confirm the replace of some files, e.g. **`avrdude.exe`** and **`avrdude.conf`**.

Download and install Zadig 2.4.

Compiler Test

At a terminal or command prompt type

```
avr-gcc --version
```

You should see a message telling you the version of the compiler installed.

ISP Programming Test

Plug in your FTDI C232HM cable into a USB port on your computer.

If you are using Windows run Zadig and select C232HM-DDHSL-0 from the drop-down list and then select the WinUSB driver.

Type (on Linux you may need to prepend **`sudo`**)

```
avrdude -c C232HM -p m644p
```

You should see the message

```
avrdude: initialization failed, rc=-1
```

Now connect the other end of your C232HM cable to your Il Matto ISP port following the instructions in section 3.14.1 of lab X2.

Now execute the avrdude command again and you should see the message

```
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.00s
avrdude: Device signature = 0x1e960a (probably m644p)
avrdude: safemode: Fuses OK (E:FF, H:9C, L:FF)
avrdude done.  Thank you.
```

USB Programming Test

Make sure you have installed the bootloader following the instructions in section 3.14.4 of lab X2.

Connect a USB cable between a USB port on your computer and the USB connector on your Il Matto board.

Ensure JP3 is in the USR position on the Il Matto.

Ensure S1 is in the USB position on the Il Matto.

Press the reset button and the LED should illuminate indicating the Il Matto is in the bootloader.

If you are using Windows run Zadig and select USBasp from the drop-down list and then select the WinUSB driver.

Type

```
avrdude -c usbasp -p m644p
```

If successful you should see the message

```
avrdude: warning: cannot set sck period. please check for usbasp firmware
update.
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.00s
avrdude: Device signature = 0x1e960a (probably m644p)
avrdude: safemode: Fuses OK (E:FF, H:9C, L:FF)
avrdude done.  Thank you.
```

Setting Up the Bitscope

Download bitscope-dso and bitscope-logic.

Run the installer for both programs.

Plug in the bitscope.

Open Device Manager → Expand "USB Serial Manager". You should see something like **FT245R**. The next part assumes you have installed the serial terminal as it uses the same drivers as the C232HM cable.

Right click and "Update Driver" → "Browse my computer for driver software" → "Let me pick from a list of available drivers on my computer" → "Have Disk" → "Browse" → Navigate to `C:\Temp\CDM v2.12.28 WHQL Certified` and select `ftdibus.inf` → "OK" → "USB Serial Port" → "Next".

It should now show up as **USB Serial Port (COMX).** Make a note of X which is the port number.

Oscilloscope (User Guide)

Start the BitScope DSO program. Click Setup and select the correct COM port for the USB interface. Select OK. Then select Power and wait for about 20 seconds!

Remove the yellow jumper from CHA and connect the AWG pin to CHA.

Select the wave button to turn on the function generator. You should see a sine wave on the scope display. Disconnect the wire from CHA to confirm you are seeing the signal.

Close the program.

Logic Analyser (User Guide)

Start the BitScope Logic program. Click Setup and select the correct COM port for the USB interface. Select OK. Then select Connect and wait for about 20 seconds!

Follow the instructions for testing the serial terminal so that you have the UART test program running on your Il Matto.

Connect your Logic Analyser to the Il Matto (L0 to PD0, L1 to PD1 and GND to G).

Set the protocol for Logic 0-3 as UART and baud rate to 9600.

Set the Display mode to Mixed for ASync 0 and ASync 1.

Set the trigger as falling edge on ASync 0.

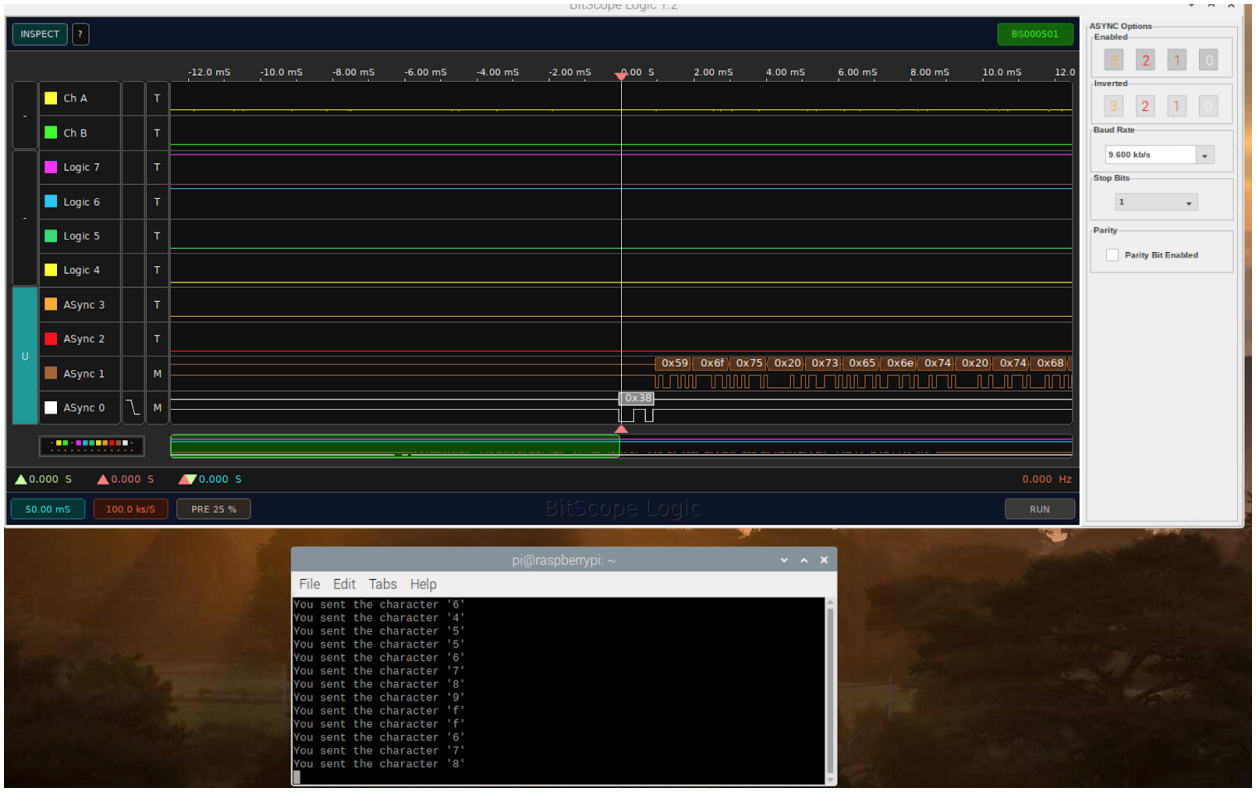
Set the Capture time to 50 ms.

Set the Sample Rate to 100 ks/s.

Set the PreTrigger to 25%.

Select Run.

Type a key and you should see something like (right click and open image in a new tab to see it at a reasonable size):



Setting Up a Serial Terminal

Download and install PuTTY to provide a serial terminal.

Download the FTDI drivers for the C232HM cable. Unzip them to a directory, e.g. **`C:\Temp\CDM v2.12.28 WHQL Certified`**.

Open Device Manager → Expand "Ports (COM & LPT)". You should see **C232HM-DDHSL-0**.

If it does not appear under Ports it should appear somewhere else in device manager. In both cases now follow the procedure below:

Right click and "Update Driver" → "Browse my computer for driver software" → "Let me pick from a list of available drivers on my computer" → "Have Disk" → "Browse" → Navigate to **`C:\Temp\CDM v2.12.28 WHQL Certified`** and select **`ftdibus.inf`** → "OK" → "USB Serial Port" → "Next".

It should now show up as **USB Serial Port (COMX).** Make a note of X which is the port number.

To configure a serial terminal, open PuTTY. Select "Connection type" as Serial. In the "Category" pane, select the last item "serial". Type the name of your port in the first box. Change the flow control in the last box to "None".

Download the UART test program, compile it and download it to your Il Matto.

Connect the TX (Orange), RX (Yellow) and GND (Black) wires from the C232HM cable to the first UART interface on the Il Matto (PD0, PD1, G).

Start your terminal and start typing and the keys should be echoed in the terminal. If this fails, check you have your TX and RX lines the right way round and that the GND cable from the C232HM is connected to your Il Matto.

X2: An introduction to Circuit Construction and Testing
https://www.youtube.com/watch?v=IpkkfK937mU&feature=emb_logo
https://www.youtube.com/watch?v=J5Sb21qbpEQ
https://www.youtube.com/watch?v=cs51fc09KzE
https://secure.ecs.soton.ac.uk/notes/ellabs/1/x2/x2.pdf
https://secure.ecs.soton.ac.uk/notes/elec1201/doc8152.pdf
https://www.engbedded.com/fusecalc/

Preparation - 25/10/20

Microcontrollers

1. What is a microcontroller?

A microcontroller is a small computer with associated peripherals all located together in one package. It is a versatile piece of electronic hardware which you can use for many different purposes by re-programming the device with a different program.

2. What is the difference between volatile and non-volatile memory?

The contents of volatile memory are lost when power is removed. Non-volatile memory contents are retained.

3. How much memory does the ATMEGA644P have for storing programs?

64K flash

4. Describe the purpose of the fuses in the microcontroller.

The fuses in the microcontroller are used to configure it and can be considered as programmable switches. Fuses are non-volatile and hence they remember their state even when power is removed from the microcontroller. There are three fuses in the ATMEGA644P with each fuse containing eight bits. These bits control the clock source (low fuse), whether parts of the device are enabled (high fuse) and what should happen if the power supply level drops (extended fuse). Details are given in table 3 including the default settings for a new AVR (the AVR supplied in your kit is a new device).

5. Calculate the new values for the low and high fuses in section 3.14.3 and 3.14.4.

Low fuse: 0xFF **High Fuse: 0x9C**

6. What is a boot loader?

A bootloader is a piece of firmware code that instructs the microcontroller how to load a new program into a separate area of memory.

7. When you have completed building your Il Matto why is it not possible to install the boot loader over the USB interface?

The USB interface on the Il Matto is a software implementation and described in the boot loader. Hence, an alternative means must be used to install it to start with.

8. Which commands do you need to execute to compile the C source files into hex files suitable for downloading to the Il Matto board?

```
avr-gcc -mmcu=atmega644p -DF_CPU=12000000 -Wall -Os prog.c -o prog.elf
avr-objcopy -O ihex prog.elf prog.hex
```

9. Which AVRDUDE command do you need to execute to download a program over the USB interface?

```
avrdude -c usbasp -p m644p -U flash:w:prog.hex
```

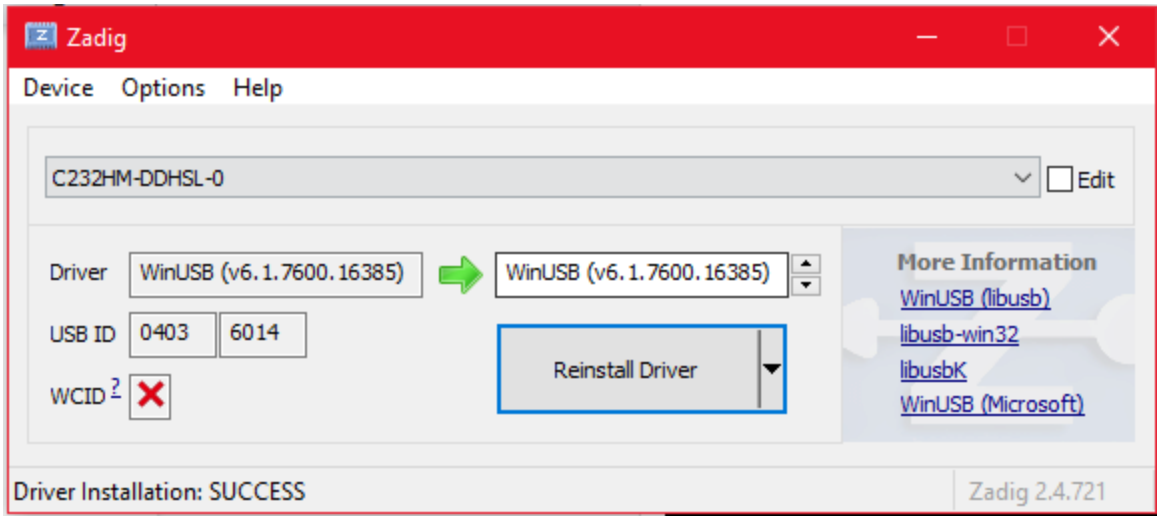
10. How many instructions per second does the microcontroller execute?

12 MIPS(for all single cycle instructions)

ISP Interface - 03/11/20

To communicate with your newly constructed embedded platform you will use the FTDI C232HM cable. This is a versatile cable and you will be using it first to communicate with the ISP port of your microcontroller to enable you to download programs and to configure the fuses.

To use the FTDI cable to program the microcontroller winAVR library must be installed so the AVR command set can be used, and the correct driver must be installed. To do this I used zadig to install WinUSB.



To connect to the microcontroller the correct programming cables must be plugged into the ISP port.

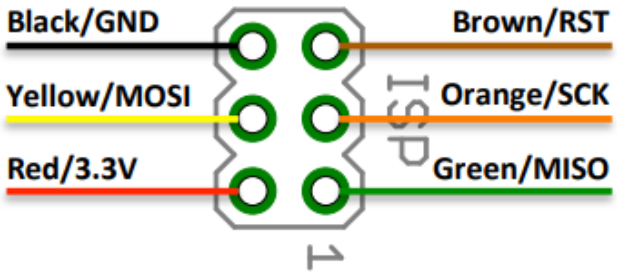


FIGURE 3: C232HM ISP connection

```
C:\Users\ludde>avr-gcc --version
avr-gcc (WinAVR 20100110) 4.3.3
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
C:\Users\ludde>avrdude -c C232HM -p m644p
avrdude: AVR device initialized and ready to accept instructions
Reading | ################################################## | 100% 0.02s
avrdude: Device signature = 0x1e960a (probably m644p)
avrdude: safemode: Fuses OK (E:FF, H:99, L:62)
avrdude done.  Thank you.
```

```
C:\XMAS revision\X2>avr-gcc -mmcu=atmega644p -DF_CPU=12000000 -Wall -Os
LEDTest.c -o LEDTest.elf
C:\XMAS revision\X2>avr-objcopy -O ihex LEDTest.elf LEDTest.hex
C:\XMAS revision\X2>avrdude -c c232hm -p m644p -U flash:w:LEDTest.hex

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.02s

avrdude: Device signature = 0x1e960a (probably m644p)
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be
performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "LEDTest.hex"
avrdude: input file LEDTest.hex auto detected as Intel Hex
avrdude: writing flash (192 bytes):

Writing | ################################################## | 100% 0.05s

avrdude: 192 bytes of flash written
avrdude: verifying flash memory against LEDTest.hex:
avrdude: load data flash data from input file LEDTest.hex:
avrdude: input file LEDTest.hex auto detected as Intel Hex
avrdude: input file LEDTest.hex contains 192 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.06s

avrdude: verifying ...
avrdude: 192 bytes of flash verified

avrdude: safemode: Fuses OK (E:FF, H:99, L:62)

avrdude done.  Thank you.
```

The fuses in the microcontroller are used to configure it and can be considered as programmable switches. Fuses are non-volatile and hence they remember their state even when power is removed from the microcontroller. There are three fuses in the ATMEGA644P with each fuse containing eight bits. These bits control the clock source (low fuse), whether parts of the device are enabled (high fuse) and what should happen if the power supply level drops (extended fuse).

```
C:\XMAS revision\X2>avrdude -c c232hm -p m644p -U lfuse:r:lf.txt:h -U
hfuse:r:hf.txt:h -U efuse:r:ef.txt:h

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.01s

avrdude: Device signature = 0x1e960a (probably m644p)
avrdude: reading lfuse memory:

Reading | ################################################## | 100% 0.00s

avrdude: writing output file "lf.txt"
avrdude: reading hfuse memory:

Reading | ################################################## | 100% 0.01s

avrdude: writing output file "hf.txt"
avrdude: reading efuse memory:

Reading | ################################################## | 100% 0.01s

avrdude: writing output file "ef.txt"

avrdude: safemode: Fuses OK (E:FF, H:99, L:62)

avrdude done.  Thank you.
```

Most of the default values configure the AVR to a suitable state. However, by default the clock source is selected to use the low precision internal RC oscillator. Your board has a high precision external crystal oscillator. Additionally you will also need to disable the “Divide Clock by 8” fuse. By changing the fuses it is also possible to install a bootloader on the device.

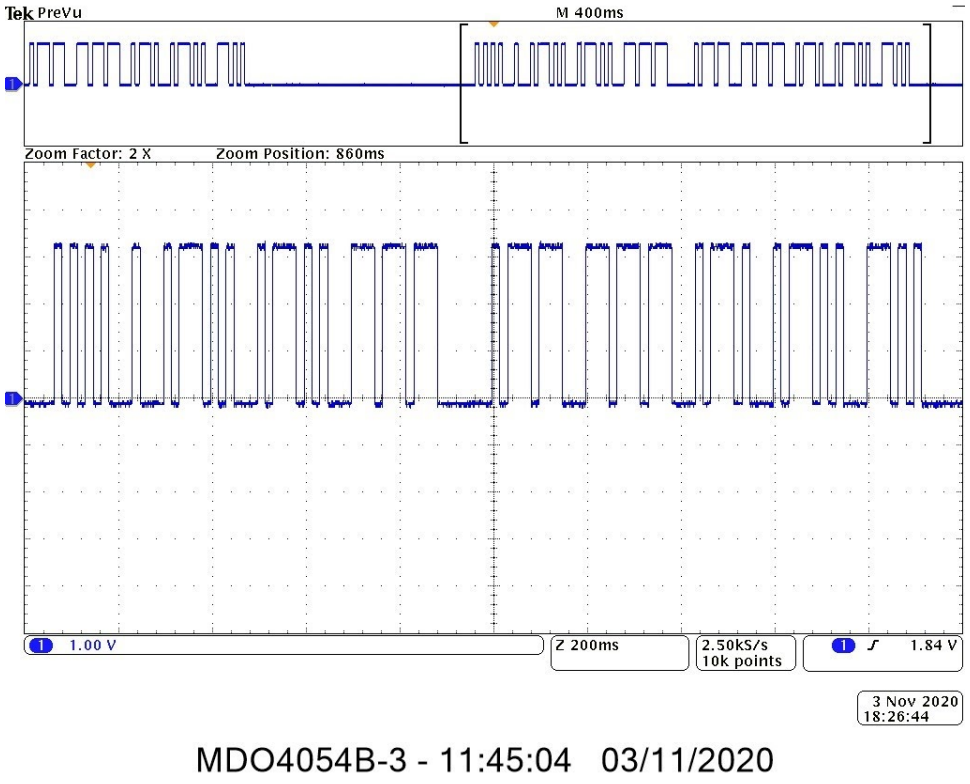
```
avrdude -c c232hm -p m644p -U lfuse:w:0xFF:m
avrdude -c c232hm -p m644p -U hfuse:w:0x9C:m
```

```
C:\XMAS revision\X2>avrdude -c c232hm -p m644p -U lfuse:r:lf.txt:h -U
hfuse:r:hf.txt:h -U efuse:r:ef.txt:h

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.02s

avrdude: Device signature = 0x1e960a (probably m644p)
avrdude: reading lfuse memory:

Reading | ################################################## | 100% 0.01s

avrdude: writing output file "lf.txt"
avrdude: reading hfuse memory:

Reading | ################################################## | 100% 0.00s

avrdude: writing output file "hf.txt"
avrdude: reading efuse memory:

Reading | ################################################## | 100% 0.01s

avrdude: writing output file "ef.txt"

avrdude: safemode: Fuses OK (E:FF, H:9C, L:FF)

avrdude done.  Thank you.
```

A bootloader can be installed onto the device to enable the device to be programmed via USB.

```
C:\XMAS revision\X2>avrdude -c c232hm -p m644p -U
flash:w:atmega644pa-12mhz_2048.hex

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.02s

avrdude: Device signature = 0x1e960a (probably m644p)
avrdude: NOTE: "flash" memory has been specified, an erase cycle will be
performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "atmega644pa-12mhz_2048.hex"
avrdude: input file atmega644pa-12mhz_2048.hex auto detected as Intel Hex
avrdude: writing flash (65536 bytes):

Writing | ################################################## | 100% 0.04s

avrdude: 65536 bytes of flash written
avrdude: verifying flash memory against atmega644pa-12mhz_2048.hex:
avrdude: load data flash data from input file atmega644pa-12mhz_2048.hex:
avrdude: input file atmega644pa-12mhz_2048.hex auto detected as Intel Hex
avrdude: input file atmega644pa-12mhz_2048.hex contains 65536 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 0.04s

avrdude: verifying ...
avrdude: 65536 bytes of flash verified

avrdude: safemode: Fuses OK (E:FF, H:9C, L:FF)

avrdude done.  Thank you.
```

Having used the boot loader to install the hello world program I am able to read the message in morse on the oscilloscope.



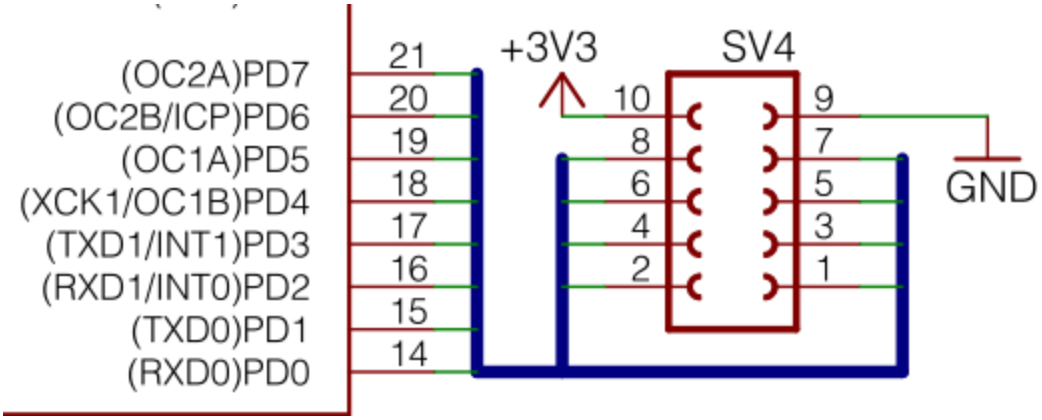
UART interface - 13/12/20

Both of the UART interfaces in the ATMEGA644P are located on port D. You will need to connect three wires from the C232HM cable (TX, RX and GND) to make the connection to the first interface on port D. You can use single strand hook-up wire to make the connection. Make sure to swap over the TX and RX lines from the cable to the board, so that the cable TX line goes to the RX line on the board and vice-versa.

A GND connection is required so that the cable and the Il Matto have a common ground. A common ground stops information being passed from device to device being read as a floating value.

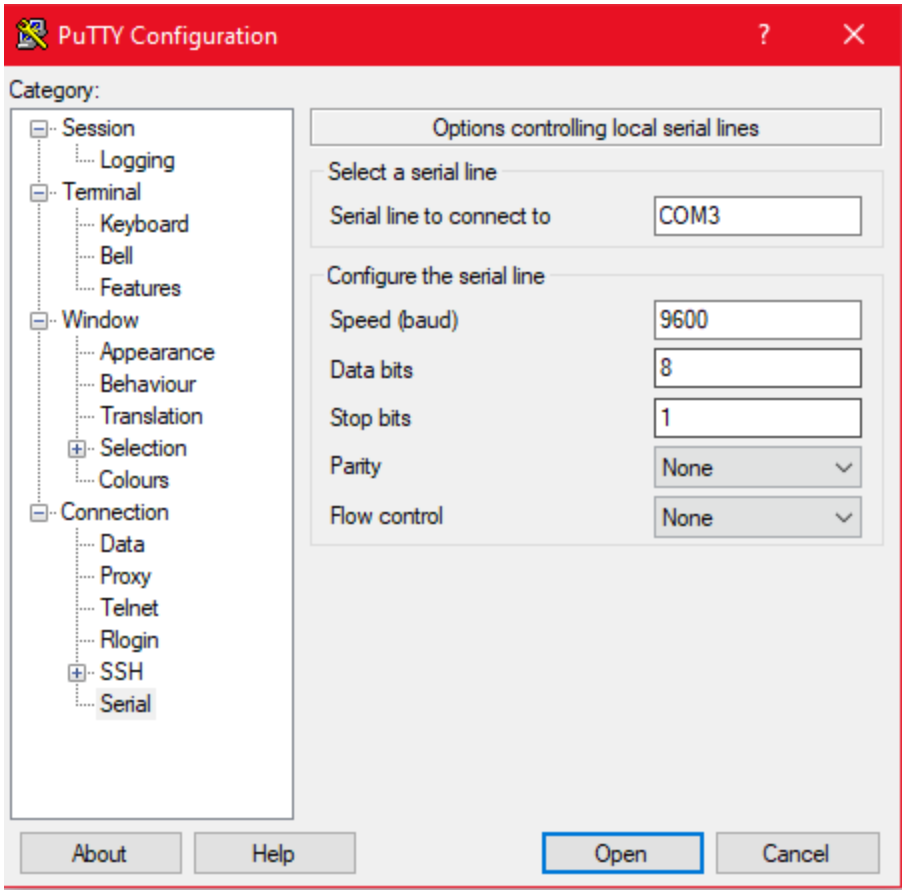
TABLE 5: FTDI C232HM cable pinout

Pin	Colour	JTAG	SPI	I²C	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



The orange wire is connected to PORTD 0.
The yellow wire is connected to PORTD 1.

Putty is used in order to communicate with the Il Matto.



This provides us with an output that shows the device is able to communicate over a serial interface.

```
You sent the character 'U'
You sent the character 'A'
You sent the character 'R'
You sent the character 'T'
```

Using a Logic analyser I am able to confirm that all the data packets are sent. This is useful for debugging a device.

GPIO interface - 13/12/20

Ensure that JP3 is in the Usr position. Build program 4 from the source file GPIOTest.c and then use AVRDUDE to download the GPIOTest.hex program to your embedded target. The LED should then extinguish and after approximately 8 seconds6 it should come back on, indicating that the program has completed and entered the boot loader. The test results can be downloaded with the command.

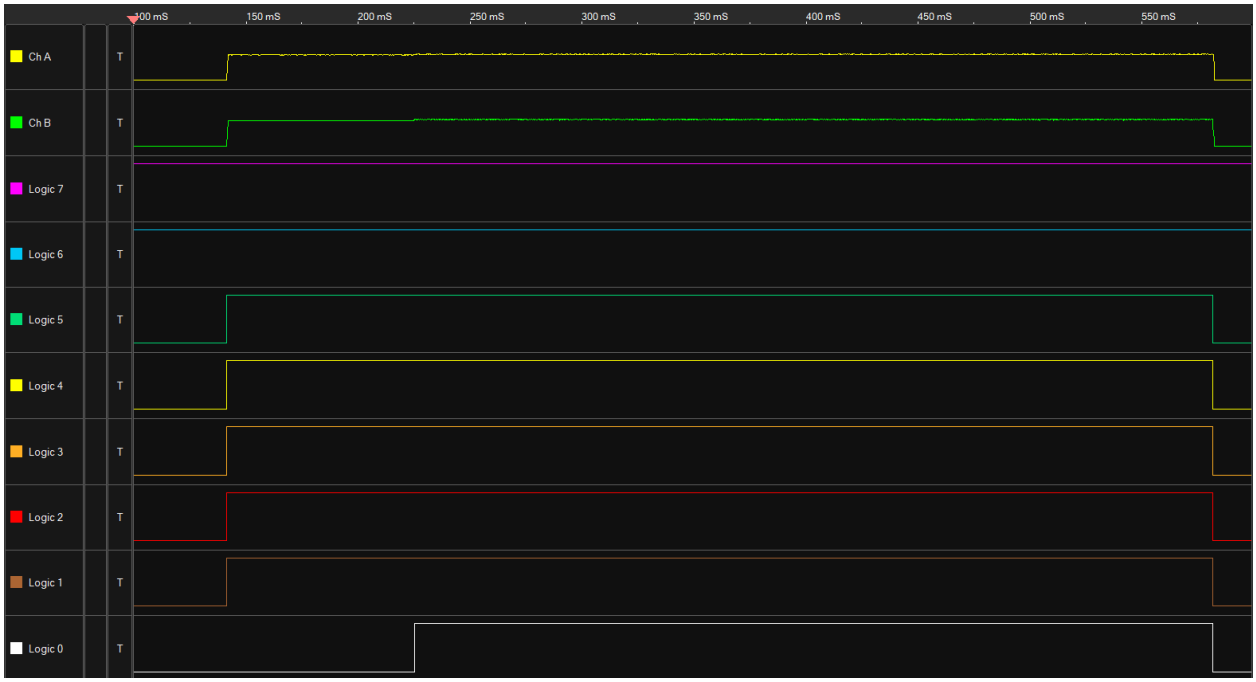
```
avrdude -c usbasp -p m644p -U eeprom:r:eeprom.txt:r
```

The output file:
eeprom.txt

```
{0000}
A->A: 0xFF 0x00 [PASS]
A->B: 0xFB 0xFB 0xFB 0xFB [FAIL]
A->C: 0xFF 0xFF 0xFF 0xFF [FAIL]
A->D: 0xFF 0xFF 0xFF 0xFF [FAIL]
B->A: 0xFF 0xFF 0xFF 0xFF [FAIL]
B->B: 0xF7 0x08 [FAIL]
B->C: 0xFF 0xFF 0xFF 0xFF [FAIL]
B->D: 0xFF 0xFF 0xFF 0xFF [FAIL]
C->A: 0xFF 0xFF 0xFF 0xFF [FAIL]
C->B: 0xFF 0xFF 0xFF 0xFF [FAIL]
C->C: 0xFF 0x00 [PASS]
C->D: 0xFF 0xFF 0xFF 0xFF [FAIL]
D->A: 0xFF 0xFF 0xFF 0xFF [FAIL]
D->B: 0xFB 0xFB 0xFB 0xFB [FAIL]
D->C: 0xFF 0xFF 0xFF 0xFF [FAIL]
D->D: 0xFF 0x00 [PASS]
```

I expect that PORTB failed due to how I had LED jumper connected.

With the signal generator connected to PORTA I am able watch the PORT as the program tests each one individually.



Connecting ports A&C and B&D together I am able to perform a loopback test to verify that aloof the pins are working correctly.
eeprom.txt

```
{0020}
A->A: 0xFF 0x00 [PASS]
A->B: 0xFB 0xFB 0xFB 0xFB [FAIL]
A->C: 0xFF 0x00 0xFF 0x00 [PASS]
A->D: 0xFB 0xFB 0xFB 0xFB [FAIL]
B->A: 0xFF 0xFF 0xFF 0xFF [FAIL]
B->B: 0xF7 0x08 [FAIL]
B->C: 0xFF 0xFF 0xFF 0xFF [FAIL]
B->D: 0xFF 0x00 0xFF 0x00 [PASS]
C->A: 0xFF 0x00 0xFF 0x00 [PASS]
C->B: 0xFF 0xFF 0xFF 0xFF [FAIL]
C->C: 0xFF 0x00 [PASS]
C->D: 0xFB 0xFB 0xFB 0xFB [FAIL]
D->A: 0xFF 0xFF 0xFF 0xFF [FAIL]
D->B: 0xFF 0x00 0xFF 0x00 [PASS]
D->C: 0xFF 0xFF 0xFF 0xFF [FAIL]
D->D: 0xE7 0x18 [FAIL]
```

C6 Digital Input & Output

https://secure.ecs.soton.ac.uk/notes/ellabs/1/c6/c6.pdf
https://blackboard.soton.ac.uk/webapps/blackboard/execute/content/file?cmd=view&content_id=_4912064_1&course_id=_190559_1
https://secure.ecs.soton.ac.uk/notes/ellabs/1/c6/SC08-11GWA.pdf
https://secure.ecs.soton.ac.uk/notes/ellabs/1/c6/pec11.pdf
https://secure.ecs.soton.ac.uk/notes/ellabs/1/c6/AST-030C0MR-R.pdf
https://en.wikipedia.org/wiki/USB#Power
https://www.dynapar.com/technology/encoder_basics/incremental_encoder/
https://www.dynapar.com/technology/encoder_basics/quadrature_encoder/

Preparation - 10/11/20

1. What is the command in C to configure a port as an output, and which header file do you need to include? 2. Calculate the value of the current limiting resistors for the seven segment display [3] to provide a current of 10mA per segment. 3. Why is it not possible to replace the seven segment display current limiting resistors with a single resistor in the common line? 4. Justify that your board can supply enough current to power the display when all segments are illuminated, when powered from a USB port. 5. Write a C array initialiser, const uint8_t segments[10] = { ... };, to store the bit patterns required to display the digits 0123456789 on the seven segment display when the segments are connected according to table 1. The relevant bit should be set if the segment is to be illuminated so that the 8-bit value can be written to the output port to display the digit. Write the values using hexadecimal notation. 6. What is the command in C to configure a port as an input? 7. When configuring a pin as an input, why is it often a good idea to enable the internal pull-up resistor? What is the value of this internal pull-up resistor? What is the command in C to enable the pull-up? 8. What voltage levels on an input pin will be read as logic High and what levels as logic Low? 9. What is switch bounce? 10. Compare and contrast one hardware and one software approach to switch de-bouncing. Digital Input & Output 5 11. With reference to the rotary encoder datasheet [1] and possibly other resources, describe how the encoder signals the direction of movement and outline how you could detect this in a program. 12. Verify that the microcontroller has enough pin drive capability to directly drive the speaker.

When working on an embedded system you should use avr libraries instead of std libraries. The library to control embedded inputs and outputs is <avr/io.h>.

To set a port to an input or output you must use the assign a value to DDR, like so:

```
DDRx = 0x00;        //Sets port “x” as inputs
PORTx = 0xFF;       //Enables inbuilt pull-up resistors

DDRy = 0xFF;        //Sets port “y” as outputs
```

Current limiting resistor:

$$V_S = V_F + V_R$$
$$V_R = V_S - V_F$$
$$V_R = 3.3 - 2.2$$
$$V_R = 1.1$$

$$V_R = IR$$
$$R = \frac{V_R}{I}$$
$$R = \frac{3.3}{10 \times 10^{-3}}$$
$$R = 110\Omega$$

It is not advised to replace the seven segment display current limiting resistors with a single resistor in the common line because every display output has a different amount of segments being lit. This means a different total voltage being provided to the display. If the display had one current limiting resistor then it will mean a different current for different numbers of segments, at best this will result in some numbers having less luminosity than others.

4. Justify that your board can supply enough current to power the display when all segments are illuminated, when powered from a USB port.

For low power devices USB can provide 100mA of current. In my X2 Lab I found that the II Matto and power LED had a current draw of 7mA.

If each segment is lit then the display will have a current draw of 8x10mA. This is below the remaining current being delivered by the USB therefore the development board is able to drive the seven segment display.

USB power standards

Specification	Current	Voltage	Power (max.)
Low-power device	100 mA	5 V[a]	0.50 W
Low-power SuperSpeed (USB 3.0) device	150 mA	5 V[a]	0.75 W
High-power device	500 mA[b]	5 V	2.5 W
High-power SuperSpeed (USB 3.0) device	900 mA[c]	5 V	4.5 W
Multi-lane SuperSpeed (USB 3.2 Gen 2) device	1.5 A[d]	5 V	7.5 W
Battery Charging (BC) 1.1	1.5 A	5 V	7.5 W
Battery Charging (BC) 1.2	5 A	5 V	25 W
USB-C	1.5 A	5 V	7.5 W
	3 A	5 V	15 W
Power Delivery 1.0 Micro-USB	3 A	20 V	60 W
Power Delivery 1.0 Type-A/B	5 A	20 V	100 W
Power Delivery 2.0/3.0 Type-C	5 A[e]	20 V	100 W

a. ^ a b The V_{BUS} supply from a low-powered hub port may drop to 4.40 V.
b. ^ Up to five unit loads; with non-SuperSpeed devices, one unit load is 100 mA.
c. ^ Up to six unit loads; with SuperSpeed devices, one unit load is 150 mA.
d. ^ Up to six unit loads; with multi-lane devices, one unit load is 250 mA.
e. ^ > 3 A (60 W) operation requires an electronically marked cable rated at 5 A.

```c
#include <avr/io.h>
#include <util/delay.h>

/*
Pin     7 6 5 4 3 2 1 0
Segment a b c d e f g dp

0       1 1 1 1 1 1 0 0     0xFC
1       0 1 1 0 0 0 0 0     0x60
2       1 1 0 1 1 0 1 0     0xDA
3       1 1 1 1 0 0 1 0     0xF2
4       0 1 1 0 0 1 1 0     0x66
5       1 0 1 1 0 1 1 0     0xB6
6       1 0 1 1 1 1 1 0     0xBE
7       1 1 1 0 0 0 0 0     0xE0
8       1 1 1 1 1 1 1 0     0xFE
9       1 1 1 0 0 1 1 0     0xE6
*/

const uint8_t segments[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE0, 0xFE, 0xE6};
```

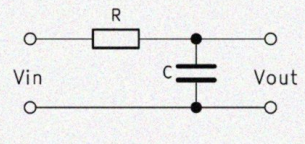
It is a good practice to enable the internal pull-up resistors (~35kΩ) to pull the input high (and not floating) when the button is not being pressed.

On the input pins voltages of less than 0.99V are read as low and voltages of more than 1.98V are read as high.

Switch bounce is noise created at the input due to the mechanical contact not instantaneously connecting.

To debounce a switch you can achieve this using hardware or software (or a combination of both).

An effective hardware solution to switch debouncing is to apply a low pass filter across the output as this blocks the high frequencies of the switch bouncing.

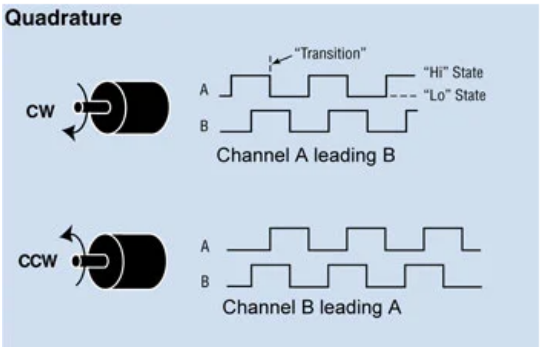


A software solution to switch debouncing is to temporarily disable the input when first activated. This will allow bounce to pass before reading from the input again.

A software solution is usually more inexpensive than a hardware solution as it does not require any additional components. However, a hardware solution is preferable when memory is at a premium.

If the switch bounce is particularly bad then both solutions may have to be used.

The rotary encoder has two output pins, they both produce a digital output when the encoder is turned. Depending on the direction that the encoder is turned then one channel will lead the other in outputting a response. From this the controller can determine which direction.



Piezo Speaker:

$$V_P = iZ$$
$$i = \frac{V_p}{Z}$$
$$i = \frac{3.3}{100}$$
$$i = 33mA$$

This is below the maximum pindrive of the Il Matto at 40mA. This means the piezo speaker can adequately drive the speaker.

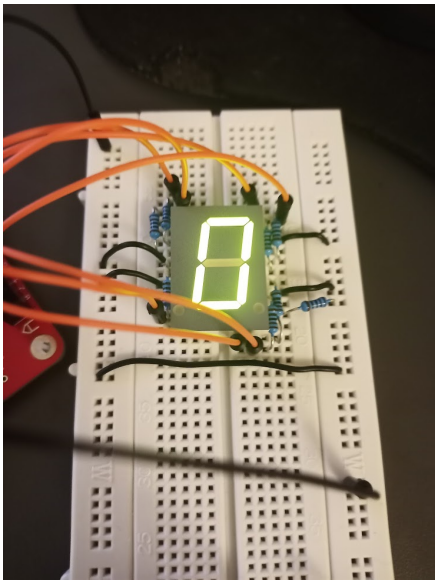
Digital Output -
Digital Input -
De-bouncing -
Rotary Encoder -
Simple Sound -
Roulette Wheel -

11/11/20

https://en.wikipedia.org/wiki/Piezoelectric_speaker

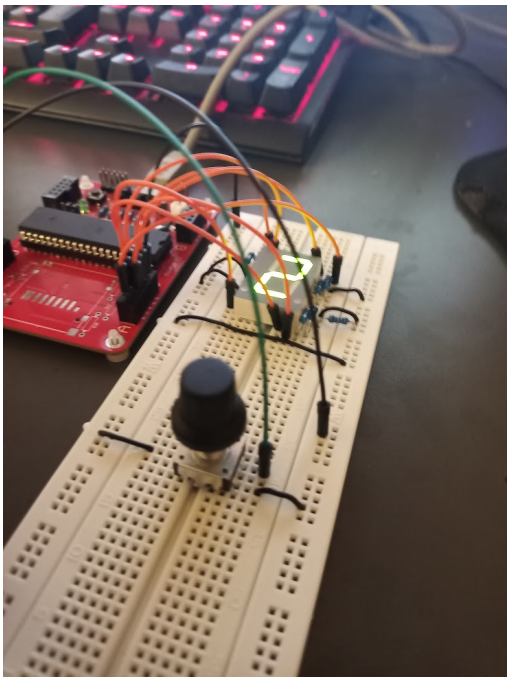
To display values on my seven segment display I wired up current limiting resistors and then connected them to the Il Matto using jumper leads. This allowed the Il Matto to drive the seven segment display.

```c
#include <avr/io.h>
#include <util/delay.h>

/*
Pin     7 6 5 4 3 2 1 0
Segment a b c d e f g dp

0       1 1 1 1 1 1 0 0     0xFC
1       0 1 1 0 0 0 0 0     0x60
2       1 1 0 1 1 0 1 0     0xDA
3       1 1 1 1 0 0 1 0     0xF2
4       0 1 1 0 0 1 1 0     0x66
5       1 0 1 1 0 1 1 0     0xB6
6       1 0 1 1 1 1 1 0     0xBE
7       1 1 1 0 0 0 0 0     0xE0
8       1 1 1 1 1 1 1 0     0xFE
9       1 1 1 0 0 1 1 0     0xE6
*/

const uint8_t segments[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE0, 0xFE, 0xE6};
uint8_t count = 0;

int main(void)
{
    DDRA = 0xFF; //Sets portA as outputs

    for (;;)
    {
        PORTA = segments[count % 10];
        count++;
        _delay_ms(1000); //Delay by 1 sec
    }
}
```



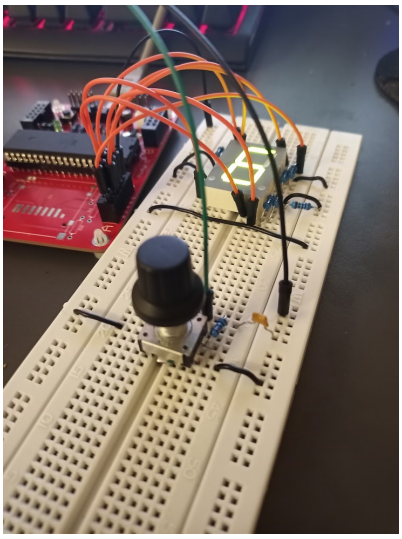
I then modified the code so that it would increment the number every time the rotary encoder was pressed. This worked for the most part but sometimes it would skip multiple places. This is likely due to switch bounce.

```c
#include <avr/io.h>
#include <util/delay.h>

/*
Pin     7 6 5 4 3 2 1 0
Segment a b c d e f g dp

0       1 1 1 1 1 1 0 0     0xFC
1       0 1 1 0 0 0 0 0     0x60
2       1 1 0 1 1 0 1 0     0xDA
3       1 1 1 1 0 0 1 0     0xF2
4       0 1 1 0 0 1 1 0     0x66
5       1 0 1 1 0 1 1 0     0xB6
6       1 0 1 1 1 1 1 0     0xBE
7       1 1 1 0 0 0 0 0     0xE0
8       1 1 1 1 1 1 1 0     0xFE
9       1 1 1 0 0 1 1 0     0xE6
*/

const uint8_t segments[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE0, 0xFE, 0xE6};
uint8_t count = 0;

int main(void)
{
    DDRA = 0xFF;  //Sets portA as outputs
    DDRC = 0x00;  //Sets portC as inputs
    PORTC = 0xFF; //Enables inbuilt pull-up resistors

    PORTA = segments[0];
    for (;;)
    {
        while ((PINC & _BV(PC7)) != 0)
        {
            count++;
        }

        while ((PINC & _BV(PC7)) == 0)
        {
            PORTA = segments[count % 10];
        }
    }
}
```



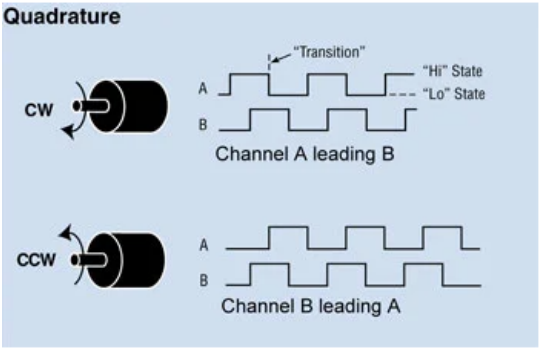
Due to the switch bounce I implemented one hardware solution, a low pass filter, and one software solution, disabling the pin. I found that this successfully debounced the pin and it would increment consistently.

```
const uint8_t segments[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE0, 0xFE, 0xE6};
uint8_t count = 0;

int main(void)
{
    DDRA = 0xFF;  //Sets portA as outputs
    DDRC = 0x00;  //Sets portC as inputs
    PORTC = 0xFF; //Enables inbuilt pull-up resistors

    PORTA = segments[0];
    for (;;)
    {
        while ((PINC & _BV(PC7)) != 0)
        {
            count++;
            _delay_ms(300);   //stop program for 300 ms to debounce switch
        }

        while ((PINC & _BV(PC7)) == 0)
        {
            PORTA = segments[count % 10];
        }

    }
}
```

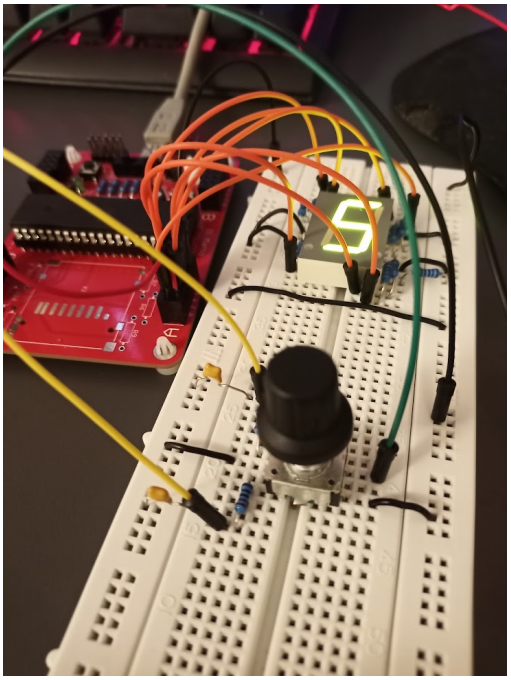


To use the rotary encoder to its full potential I used the I used the quadrature signal to increment and decrement the count when it detected a rotation.To implement this I detected a rising edge trigger while another signal is true for each of the directions.

Reading of the rotary encoder was temperamental even after implementing debouncing techniques. My implementation works but not very successfully.



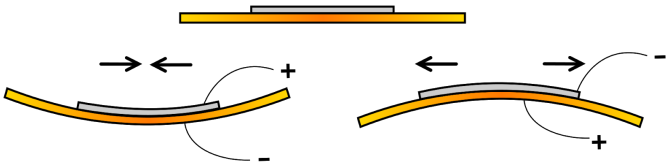
```
int main(void)
{
    DDRA = 0xFF;  //Sets portA as outputs
    DDRC = 0x00;  //Sets portC as inputs
    PORTC = 0xFF; //Enables inbuilt pull-up resistors

    PORTA = segments[0];

    for (;;)
    {
        while ((PINC & _BV(PC0)) == 0)
        {
            nowA = (PINC & _BV(PC1)); // rising edge detection
            if (nowA != lastA)        //by ka7ehk on avrfreaks.net
            {
                if (nowA > 0)
                {
                    count++;
                    _delay_ms(300); //stop program for 10 ms to debounce switch
                }
            }
        }
        while ((PINC & _BV(PC1)) == 0)
        {
            nowB = (PINC & _BV(PC0)); // rising edge detction
            if (nowB != lastB)        //by ka7ehk on avrfreaks.net
            {
                if (nowB > 0)
                {
                    count--;
                    _delay_ms(300); //stop program for 10 ms to debounce switch
                }
            }
        }

        PORTA = segments[count % 10];
    }
}
```



After speaking to my instructor I was informed of a better method that would look at a change in both the A signal and the B signal and so would operate more reliably. I rewrote my code to implement this changed technique and found that it would work reliably.

```
int main(void)
{
    uint8_t count = 0;
    uint8_t lastAB = 0x0, AB = 0x0;

    DDRA = 0xFF;  //Sets portA as outputs
    DDRC = 0x00;  //Sets portC as inputs
    PORTC = 0xFF; //Enables inbuilt pull-up resistors

    PORTA = segments[0];

    for (;;)
    {
        AB = rot_AB(PINC); // sample encoder

        if ((AB == 0x0 && lastAB == 0x2) || (AB == 0x3 && lastAB == 0x1)) // if CW
        {
            count--;
        }

        if ((AB == 0x3 && lastAB == 0x2) || (AB == 0x0 && lastAB == 0x1)) // if CCW
        {
            count++;
        }

        PORTA = segments[count % 10]; //display the  value
        _delay_ms(10); // debounce

        lastAB = AB; //update encoder position
    }
}
```

The piezo electric buzzer works by oscillating a crystal lattice. When a positive voltage it flexes in one direction and when a negative voltage is applied it flexes in the opposite direction. This means that to create a sound I have to alternate positive and negative voltages.



To test this I first created a simple program that creates a tone, and then implemented this into the button pressing program.

```c
int main(void)
{
    DDRA = 0xFF;  //Sets portA as outputs
    DDRC = 0x00;  //Sets portC as inputs
    PORTC = 0xFF; //Enables inbuilt pull-up resistors

    PORTA = segments[0];
    for (;;)
    {
     PORTA &= ~_BV(PA0); //Set bit n of port x low
     _delay_ms(1);
     PORTA |= _BV(PA0); //Set bit n of port x high
     _delay_ms(1);

    }
}
```

```c
int main(void)
{
    DDRA = 0xFF;  //Sets portA as outputs
    DDRC = 0x00;  //Sets portC as inputs
    PORTC = 0xFF; //Enables inbuilt pull-up resistors

    PORTA = segments[0];
    for (;;)
    {
        while ((PINC & _BV(PC7)) == 0)
        {
            count++;
            uint8_t i;
            for (i = 0; i < 150; i++)
            {
                PORTA &= ~_BV(PA0); //Set bit n of port x low
                _delay_ms(1);
                PORTA |= _BV(PA0); //Set bit n of port x high
                _delay_ms(1);
            }
        }

        while ((PINC & _BV(PC7)) != 0)
        {

            PORTA = segments[count % 10];
        }
    }
}
```

I then implemented it on my rotary encoder, to create a brief click when a value is updated.

```c
uint8_t buzz(uint8_t length)
{
    uint8_t i;
       for (i = 0; i < length; i++)
       {
           PORTA &= ~_BV(PA0); //Set bit n of port x low
           _delay_ms(0.5);
           PORTA |= _BV(PA0); //Set bit n of port x high
           _delay_ms(0.5);
       }
    return 0;
}

int main(void)
{
    uint8_t count = 0;
    uint8_t lastAB = 0x0, AB = 0x0;

    DDRA = 0xFF;  //Sets portA as outputs
    DDRC = 0x00;  //Sets portC as inputs
    PORTC = 0xFF; //Enables inbuilt pull-up resistors

    PORTA = segments[0];

    for (;;)
    {
        AB = rot_AB(PINC); // sample encoder

        if ((AB == 0x0 && lastAB == 0x2) || (AB == 0x3 && lastAB == 0x1)) // if CW
        {
            count--;
            buzz(30);
        }

        if ((AB == 0x3 && lastAB == 0x2) || (AB == 0x0 && lastAB == 0x1)) // if CCW
        {
            count++;
            buzz(30);
        }
```

C7

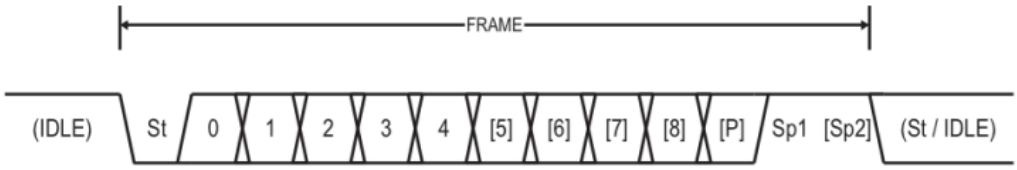
Preparation -

https://blackboard.soton.ac.uk/webapps/blackboard/execute/content/file?cmd=view&content_id=_4922450_1&course_id=_190559_1

U(S)ART stands for - Universal (synchronous) asynchronous receiver-transmitter.

UART supports upper to 2 MBaud for serial communication.

Figure 18-4. Frame Formats



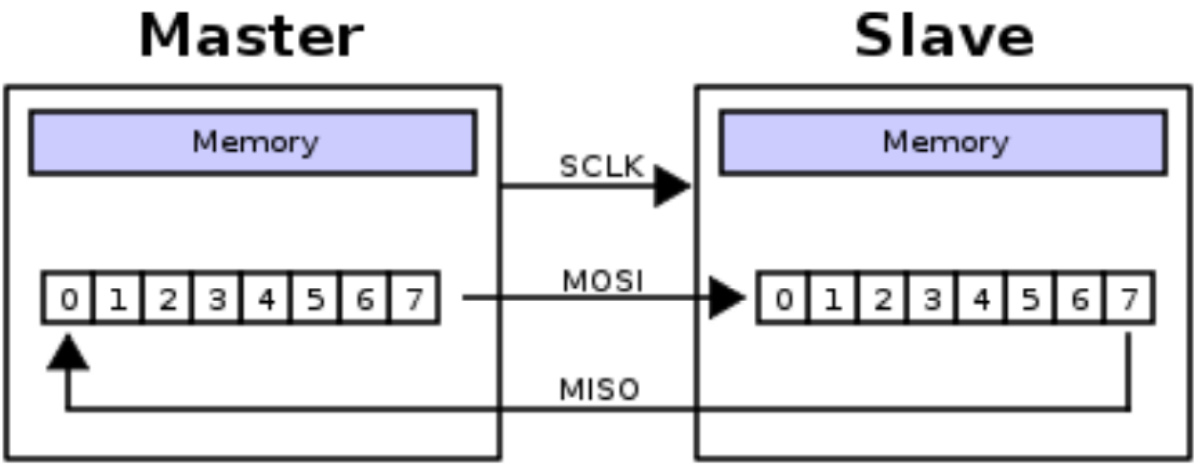
St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxDn or TxDn). An IDLE line must be high.

When an 8-bit ASCII character is transmitted over the UART at least 10 bits are sent, but with parity and a second stop bit then it can be as high as 12.

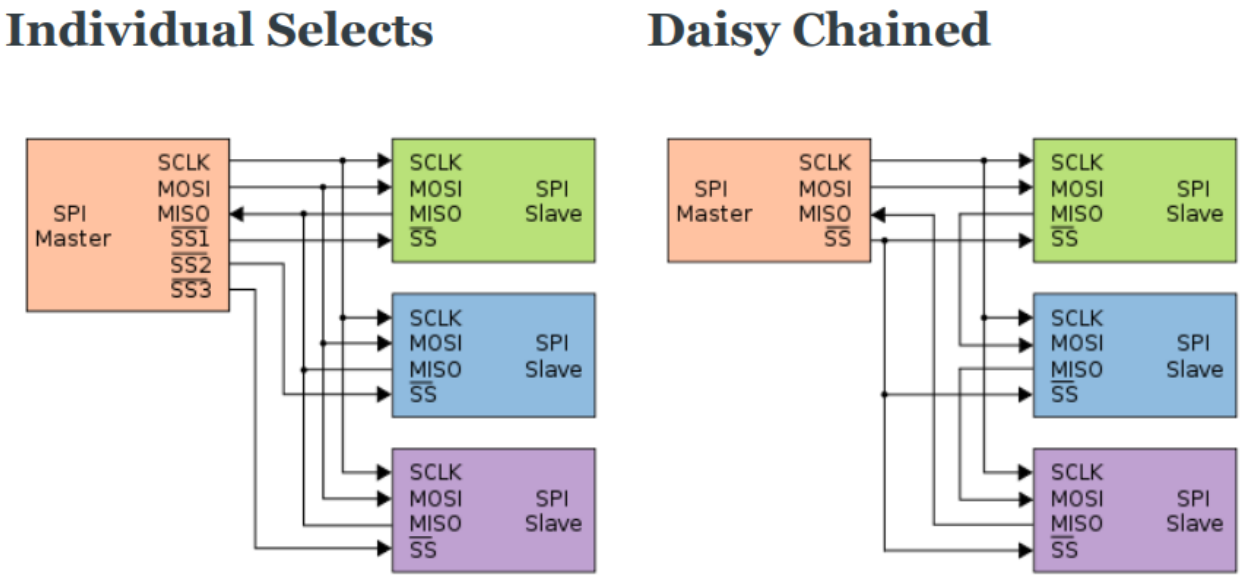
UART can only connect a master and slave together.

SPI stands for - Serial Peripheral Interface.

SPI supports upper to 100 MBaud for serial communication.



When an 8-bit ASCII character is transmitted over the SPI 8 bits are sent over the MOSI line, the ~SS pin must also be selected for the slave to receive the information.



In individual select mode as many slaves can be connected as the number of ~SS pins the master has. In daisy chained mode any number of slaves can be connected. This is functionally limited to 20 slaves because of signal degradation on the bus.

I2C stands for - Inter-Integrated Circuit
I2C is called TWI (Two Wire Interface) in avrdude.

I2C supports upper to 5 MBaud for serial communication.

When an 8-bit ASCII character is transmitted over the I2C it follows the following series of operations.

– send a start bit,
– send the 7-bit slave address it wishes to talk to,
– send a direction bit (write(0), read(1))
– wait for/send an acknowledge bit
– send/receive data byte (8 bits)
– wait for/send an acknowledge bit
– send the stop bit

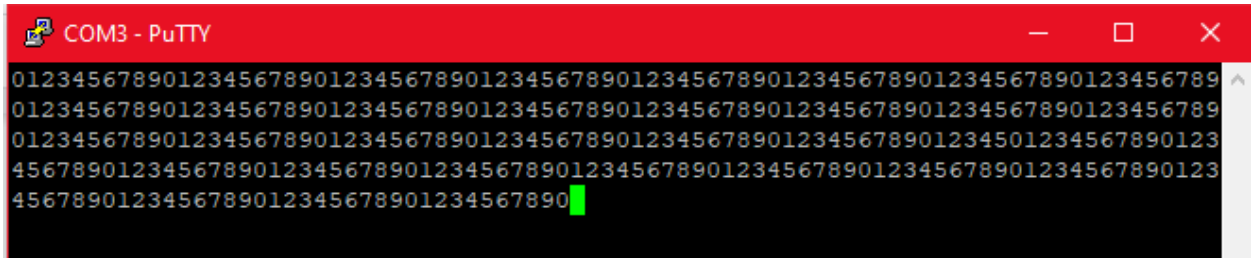
Meaning to send 8 bits of information 18 bits are sent and 2 acknowledgement bits are received.

I2C can support up to 128 simultaneous devices, these can be any number of master or slave devices.

18/11/20

I modified my code from the first part of the C6 Lab, so that it would also print the value to the terminal. This allowed me to see what state the program was in.

```
22  const uint8_t segments[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE0, 0xFE, 0xE6};
23  uint8_t count = 0;
24
25  int main(void)
26  {
27      DDRA = 0xFF; //Sets portA as outputs
28      init_debug_uart0();
29
30      for (;;)
31      {
32          int display = count % 10;
33          printf("%i", display);
34          PORTA = segments[display];
35          count++;
36          _delay_ms(1000); //Delay by 1 sec
37      }
38  }
```



I then made it so that it printed a message every time there was a count overflow and then entered a new line.

```
22  const uint8_t segments[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE0, 0xFE, 0xE6};
23  uint8_t count = 0;
24
25  int main(void)
26  {
27      DDRA = 0xFF; //Sets portA as outputs
28      init_debug_uart0();
29
30      for (;;)
31      {
32          int display = count % 10;
33          printf("%i", display);
34          while (display == 0)
35          {
36              printf("\n");
37              fprintf(stderr, "Count overflow\n\r");
38              break;
39          }
40          PORTA = segments[display];
41          count++;
42          _delay_ms(1000); //Delay by 1 sec
43      }
44  }
```

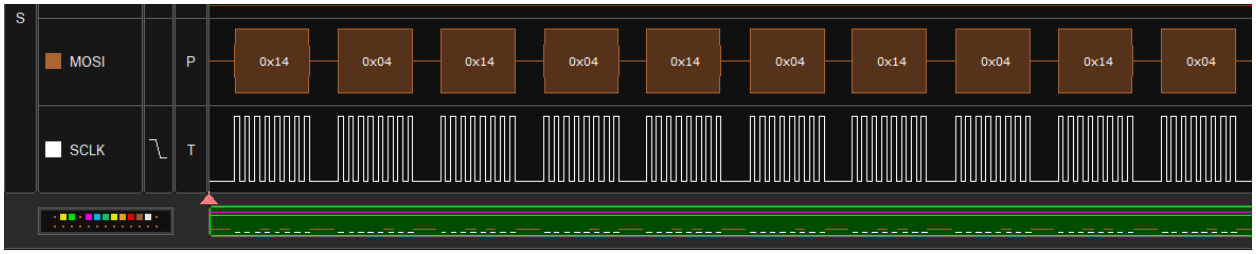
I then made it so that it would wait until an input was made from the terminal before it began counting again.

```c
const uint8_t segments[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE0, 0xFE, 0xE6};
uint8_t count = 0;

int main(void)
{
    DDRA = 0xFF; //Sets portA as outputs
    init_debug_uart0();

    for (;;)
    {
        int display = count % 10;
        printf("%i", display);
        while (display == 9)
        {
            char str1[1];
            printf("\n");
            fprintf(stderr, "Count overflow\n\r");
            while (fscanf(stdin, "%s", str1) == 0)
            {
            }
            break;
        }
        PORTA = segments[display];
        count++;
        _delay_ms(1000); //Delay by 1 sec
    }
}
```

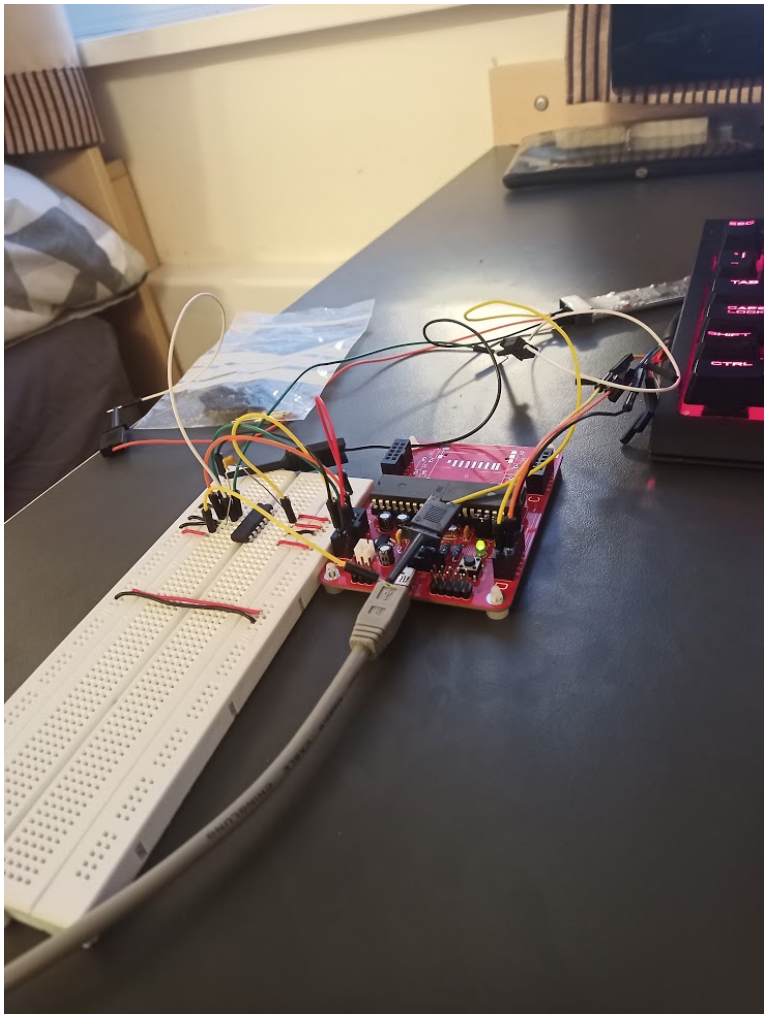
```c
#include <avr/io.h>
#include "spi.h"
#include "digitalPot.h"

int main(void)
{
    uint8_t i;
    init_pot();

    for(;;)
    {
        setXpot(0x00);
        setYpot(0x00);

        for(i=0; i <= 255; i++)
        {
            incXpot();
            incYpot();
        }
    }
}
```

```c
#include "spi.h"

void init_pot(void)
{
    /* TODO: Code to initiliase interface */
    init_spi_master();
}

void decXpot(void)
{
    /* TODO: Code to decrement first pot by 1 */
    uint8_t decx = 0x08;    //0000 1000
    spi_tx(decx);
    spi_rx();
}

void incXpot(void) ···

void decYpot(void) ···

void incYpot(void)
{
    /* TODO: Code to increment second pot by 1 */
    uint8_t incy = 0x14;    //0001 0100
    spi_tx(incy);
    spi_rx();
}

void setXpot(uint8_t x)
{
    /* TODO: Code to set first pot to a value between 0 and 255 */
    uint8_t setx = 0x00;    //0000 00nn nnnn nnnn
    spi_tx(setx);
    spi_rx();
    spi_tx(x);
    spi_rx();
}
```

To test the digital potentiometer I connected my logic analyse to the clock and MOSI wires and read the data packages being sent. These are what I expected as they are the same values as incXpot() and incYpot().



My connected digital potentiometer.



C8

Preparation -

Timers/Counters and Pulse-Width Modulation

8-bit resolution is not enough to cover the audio range with the resolution that would match the musical note. Hence, I will use timer 1, a 16-bit timer.

TTCR1A								
Bit	7	6	5	4	3	2	1	0
(0x80)	COM1A1	COM1A0	COM1B2	COM1B0	-	-	WGM11	WGM10
R/W	R/W	R/W	R/W	R/W	R	R	R/W	R/W
Initial Val	0	0	0	0	0	0	0	0
Set Val	0	1	0	0	0	0	0	1
TTCR1B								
Bit	7	6	5	4	3	2	1	0
(0x81)	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Initial Val	0	0	0	0	0	0	0	0
Set Val	0	0	0	1	0	0	1	0

Selecting a pre-scalar:

$$f_{oc} = \frac{f_{clk\text{-}I/O}}{2 \times N_{max} \times TOP}$$

$$N_{max} = \frac{f_{clk\text{-}I/O}}{2 \times f_{oc} \times TOP}$$

$$N_{max} = \frac{12000000}{2 \times 18000 \times 1}$$

$$N_{max} = 333$$

Most timers pre-scalars {1, 8, 64, 256, or 1024} would work. I will be using a pre-scalar of 8 in my code.

```
32  void init_tone(void)
33  {
34      DDRD |= _BV(PD5);     /* enable ouput driver for OC1A */
35      TCCR1A = _BV(COM1A0) /* toggle OC1A on match */
36             | _BV(WGM10);  /* frequency (f) correct PWM, */
37      TCCR1B = _BV(WGM13)  /* varying f with OCR1A */
38             | _BV(CS11);   /* prescaler set to 8 */
39  }
```

```
void tone(uint16_t frequency)
{
   /*
   We rely on the compiler to substitute the constant part of the expression.
   1/2 for symmetric PWM & 1/2 for toggle output
   */
   OCR1A = (uint16_t)(F_CPU / (2 * 2 * TONE_PRESCALER) / frequency);
}
```

I will be using Timer2 for my amplitude control, as one of the two outputs for Timer0 is used with the usb interface.

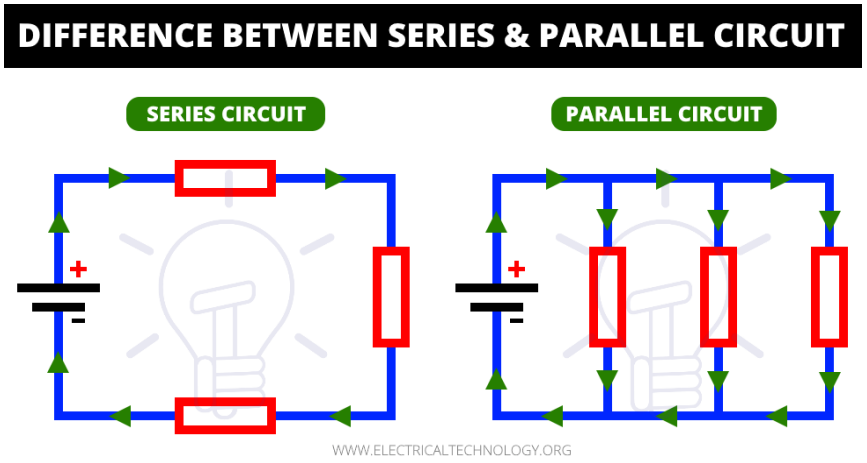
Maximum frequency at 8-bit resolution:

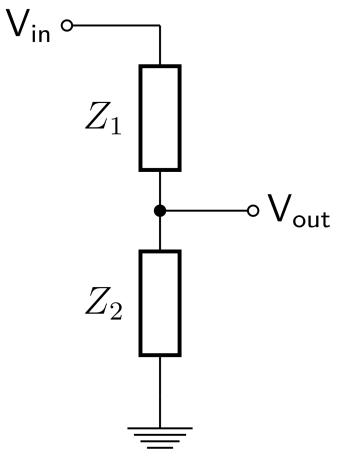
$$f_{oc} = \frac{f_{clk-I/O}}{N \times 256}$$
$$f_{oc} = \frac{12000000}{1 \times 256}$$
$$f_{oc} = 46.9KHz$$

TTCR2A								
Bit	7	6	5	4	3	2	1	0
(0xB0)	COM2A1	COM2A0	COM2B2	COM2B0	-	-	WGM21	WGM20
R/W	R/W	R/W	R/W	R/W	R	R	R/W	R/W
Initial Val	0	0	0	0	0	0	0	0
Set Val	1	0	0	0	0	0	1	1
TTCR2B								
Bit	7	6	5	4	3	2	1	0
(0xB1)	FOC2A	FOC2B	-	-	WGM22	CS22	CS21	CS20
R/W	W	W	R	R	R/W	R/W	R/W	R/W
Initial Val	0	0	0	0	0	0	0	0
Set Val	0	0	0	0	1	0	0	1

```
116  void init_volume(void)
117  {
118      DDRD |= _BV(PD7);     /* enable ouput driver for OC1A */
119      TCCR1A = _BV(COM2A0) /* toggle OC2A on match */
120           | _BV(WGM21);    /* fast PWM, */
121           | _BV(WGM20);    /* fast PWM, */
122      TCCR1B = _BV(WGM22)  /* varying f with OCR2A */
123           | _BV(CS20);     /* prescaler set to 1 */
124  }
```

Signal	Port	Pin Number
Tone	D	5
Volume	D	7





Using the oscilloscope you can use the time base to find the time period of the signal, and then use f = 1/T to find the frequency.

Port Registers ($x \in$ {A,B,C,D})

PINx, PORTx, DDRx	Input, Output and Direction registers

Bit Manipulation ($n \in$ {0,1,...,7}, $r \in$ {I/O Registers})

uint8_t value;	Declare value as an 8-bit byte
#define _BV(n) (1 << (n))	Bit Value
value = 0xFF;	Set all 8-bits of byte value
value = 0x00;	Clear all 8-bits of byte value
value = ~value;	Invert all bits of byte value
value |= _BV(n);	Set bit n of byte value
value &= ~_BV(n);	Clear bit n of byte value
if bit_is_set(r, n) { ... }	Test if bit n of r is set
if bit_is_clear(r, n) { ... }	Test if bit n of r is clear
loop_until_bit_is_set(r, n);	Wait until bit n of r is set
loop_until_bit_is_clear(r, n);	Wait until bit n of r is clear

Input

DDRx = 0x00;	Set 8-bits of port x as inputs
PORTx = 0xFF;	Enable pull-ups on input port x
PORTx = 0x00;	Configure inputs as tri-state on port x
value = PINx;	Read value of port x
DDRx &= ~_BV(n);	Set bit n of port x as input
PORTx |= _BV(n);	Enable pull-up on bit n of port x
PORTx &= ~_BV(n);	Configure tri-state on bit n of port x
if (PINx & _BV(n)) { ... }	Test value of pin n on port x

Output

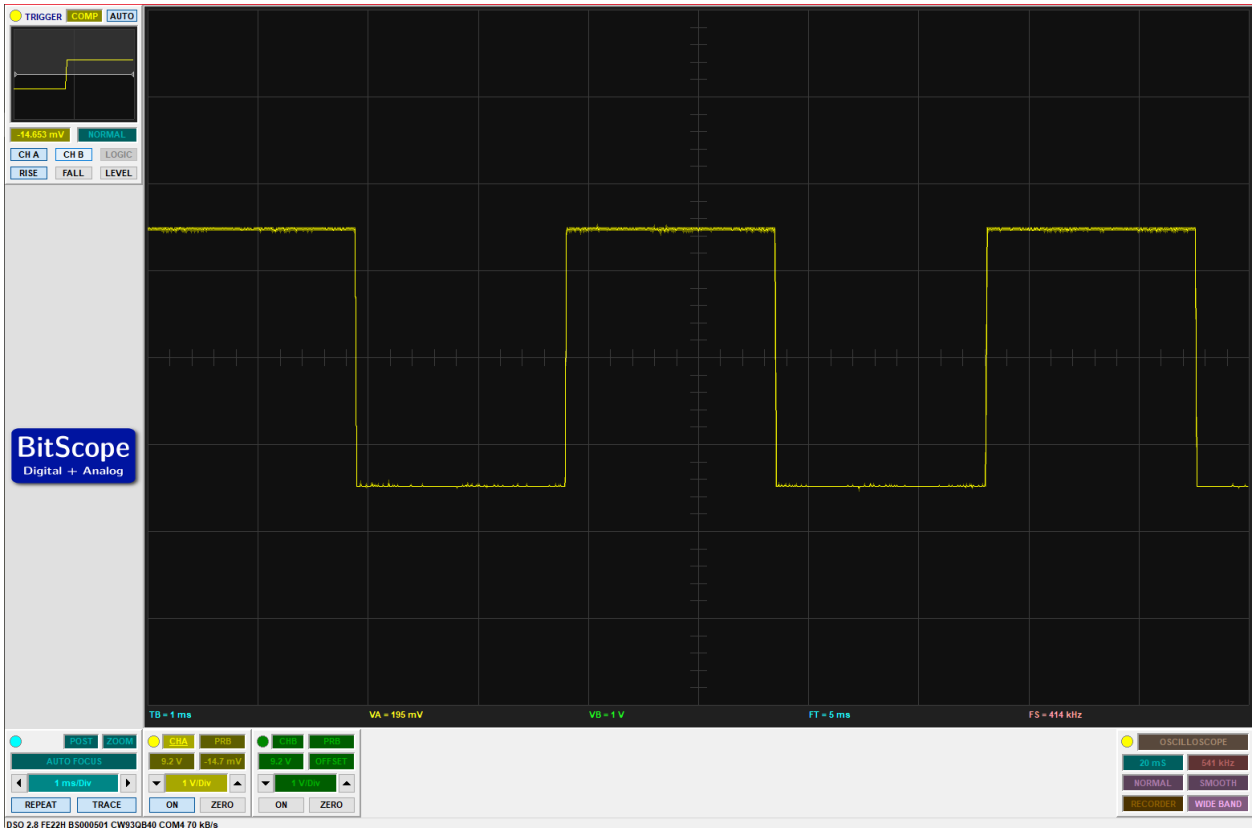
DDRx = 0xFF;	Set 8-bits of port x as outputs
PORTx = 0xFF;	Set all output bits on port x high
PORTx = 0x00;	Set all output bits on port x low
PINx = 0xFF;	Toggle all output bits on port x high
DDRx |= _BV(n);	Set bit n of port x as output
PORTx |= _BV(n);	Set bit n of port x high
PORTx &= ~_BV(n);	Set bit n of port x low
PINx |= _BV(n);	Toggle bit n of port x

Input/Output

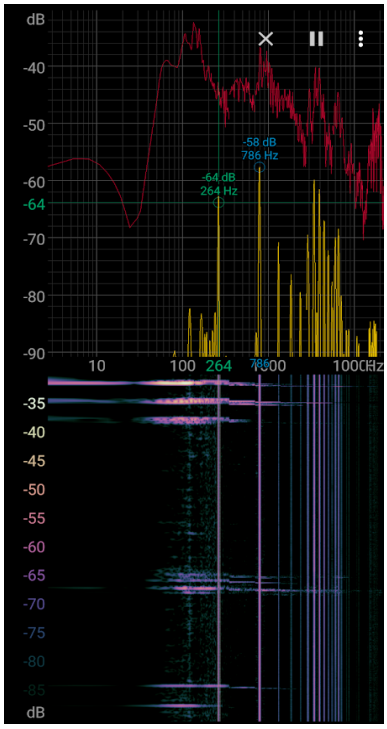
DDRx = 0x0F;	High 4-bits input, low 4-bits output
PORTx |= 0x0F;	Set all output bits high
PORTx &= 0xF0;	Set all output bits low
value = PINx & 0xF0;	Read input bits on port x

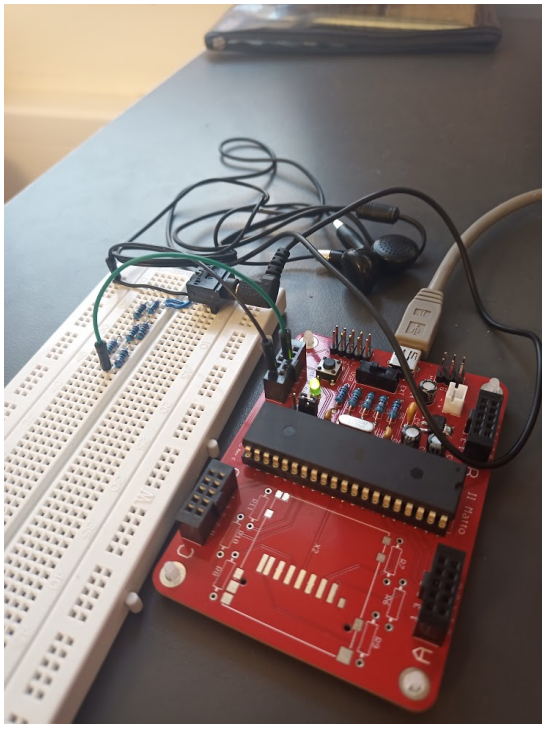
25/11/20

Setting my defined tone to 262 Hz, a middle C tone. I connected my bitscope to the output at pin D5 (pin for Timer1). I measured the output frequency as a clear square wave.



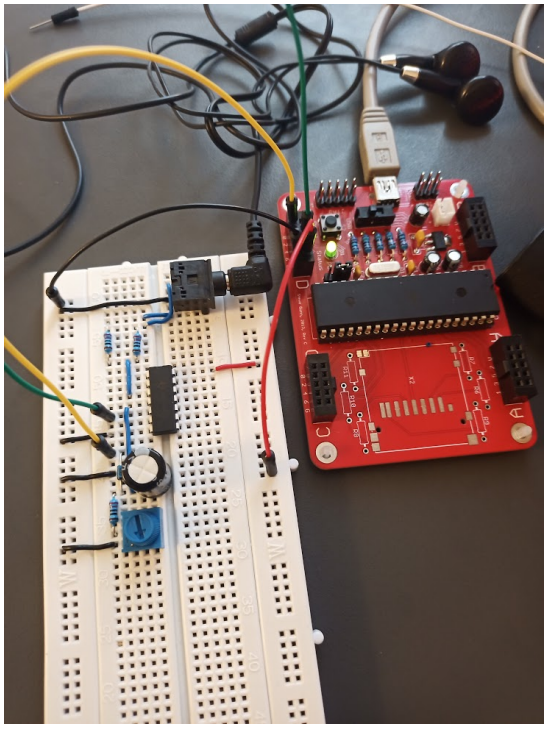
$$f = \frac{1}{T}$$
$$f = \frac{1}{3.9\times10^{-3}}$$
$$f = 256.4\ Hz$$

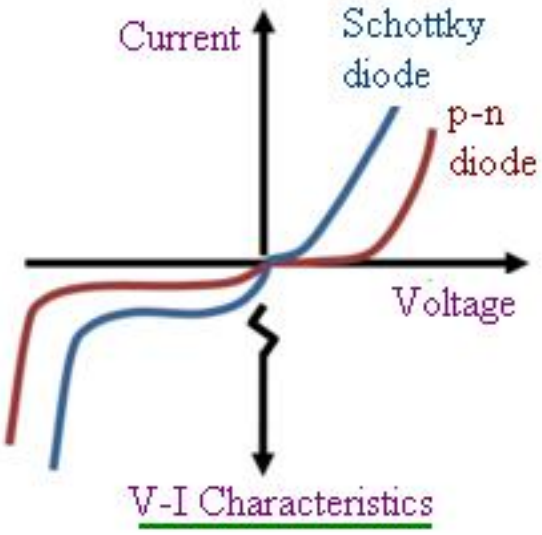




I then modified the main function of my code so that it would scale up and down the frequency range in “et_scale.h”.

```
int main(void)
{
   init_tone();

   for (;;)
   {
      int i;
      for (i = 0; i <= ET_SCALE_TOP; i++)
      {
         tone(et_scale[i]);
         _delay_ms (100);
      }
      for (i = ET_SCALE_TOP; i >= 0; i--)
      {
         tone(et_scale[i]);
         _delay_ms (100);
      }
   }
}
```





The schottky diode used on the trigger will limit the current supplied to the capacitor and then the capacitor will slowly discharge over the duration of the note. This creates a decaying amplitude which will make the voltage supplied to the headphones more like the wave packet when an instrument creates sound.

Making the pulse time significantly shorter than the length of the note makes the note cut off, making the overall sound more electronic.

I then modified the code so that it would briefly pulse a trigger. This charged the capacitor and let it decay over the note to give it a more natural sound.

```
int main()
{
    uint16_t f;

    DDRB |= _BV(PB7); /* LED */
    DDRD |= _BV(PD0); /* Trigger*/
    init_tone();

    for (;;)
    {
        melody2freq(melody); /* initialise  */
        while ((f = melody2freq(NULL)) != M2F_END)
```

```
        {
            if (f == M2F_UNKOWN)
            {
                continue; /* skip unknown symbols */
            }

            tone(f);
            PORTD |= _BV(PD0);      /* set Trigger */
            _delay_ms(10);
            PORTD &= _BV(PD0);      /* Clear Trigger  */
            _delay_ms(STEP_DELAY_MS);
            PORTB ^= _BV(PB7);      /* toggle LED */

        }
        _delay_ms(STEP_DELAY_MS);
        _delay_ms(STEP_DELAY_MS);
    }
}
```

I then added a volume control using a second PWM pulse to control the analog switch chip.

```
void init_volume(void)
{
    DDRD |= _BV(PD6);
    DDRD |= _BV(PD7);

    TCCR2A = _BV(COM2A1) /* toggle OC2A on match */
         | _BV(WGM21)    /* fast PWM, */
         | _BV(WGM20);   /* fast PWM, */
    TCCR2B = _BV(CS20);  /* prescaler set to 1 */
}

void volume(uint8_t x)
{
    uint8_t vol = (256*(x/100));
    OCR2A = vol;
}
```

I finally modified my code so that it would increment the volume every time a note was played. Once the volume had reached max it would cause an overflow and the volume would start counting from zero again.

```
uint8_t vol = 1;

    for (;;)
    {
        melody2freq(melody); /* initialise  */
        while ((f = melody2freq(NULL)) != M2F_END)
        {
            if (f == M2F_UNKOWN)
            {
                continue; /* skip unknown symnols */
            }

            tone(f);
            volume(vol);
            _delay_ms(STEP_DELAY_MS);
            PORTB ^= _BV(PB7);      /* toggle LED */

        }
        _delay_ms(STEP_DELAY_MS);
        _delay_ms(STEP_DELAY_MS);

        vol++;
    }
```

C9

Preparation -

30/11/20

Analogue Input

- The resolution of the ADC is 10 bits.
- There are 8 multiplexed input channels that are single ended inputs and they are found on the A Port.
- A channel is selected for single ended input by:

```
ADMUX = n; /* Select channel n */
```

If the ADC is configured for single ended input and returns a result of 0x00FF what is the voltage that was measured?

(255 /1023)*3.3V = 0.823V

- The Il Matto has different ADC pre-scalars of 2, 4, 8, 16, 32, 64, 128
- 13 clock cycles are required to capture a measurement on a single ended input.
- The fastest prescaler that can be used whilst maintaining full precision is 64.

Write a C statement to configure the pre-scaler setting calculated above and enable the ADC.

```
void init_adc(void)
{
    ADCSRA |= _BV(ADPS2) | _BV(ADPS1);  // F_ADC = F_CPU/64
    ADCSRA |= _BV(ADEN);                // Enable ADC
}
```

Write a C statement that starts a single conversion.

- ADCSRA |= _BV(ADSC);

Write a C statement that waits until a conversion is complete.

- loop_until_bit_is_set(ADIF, ADCSRA); //Wait until bit ADIF of ADCSRA is set

- When Auto Triggering is used, the prescaler is reset when the trigger event occurs. This assures a fixed delay from the trigger event to the start of conversion. In this mode, the sample-and-hold takes place 2 ADC clock cycles after the rising edge on the trigger source signal. Three additional CPU clock cycles are used for synchronization logic.
- In Free Running mode, a new conversion will be started immediately after the conversion completes, while ADSC remains high.

02/12/20

```
#define V_MAX 3.3

void init_adc(void)
{
    ADCSRA |= _BV(ADPS2) | _BV(ADPS1); // F_ADC = F_CPU/64
    ADCSRA |= _BV(ADEN);               // Enable ADC
}

uint16_t read_adc(void)
{
    ADCSRA |= _BV(ADSC);            // Start ADC conversion
    while (ADCSRA & _BV(ADSC));     //wait until result is valid
    return ADC;
}

int main(void)
{
    uint16_t result;
    double voltage;

    init_debug_uart0();
    init_adc();

    for (;;)
    {
        result = read_adc();            //get position of an ADC
        voltage = V_MAX / 1023 * result;    //calculate the voltage
        printf("%4d : %6.5lfV\n", result, voltage);
        _delay_ms(1000);
    }
}
```

Initially I could not quite get the full voltage range for my potentiometre I found that it went away when I connected the wires closer to the potentiometer so I expect this is because it is due to spurious capacitance or inductance.

$$noise = log2 \frac{3.3}{V max - V min}$$
$$noise = log2 \frac{3.3}{3.3 - 0}$$

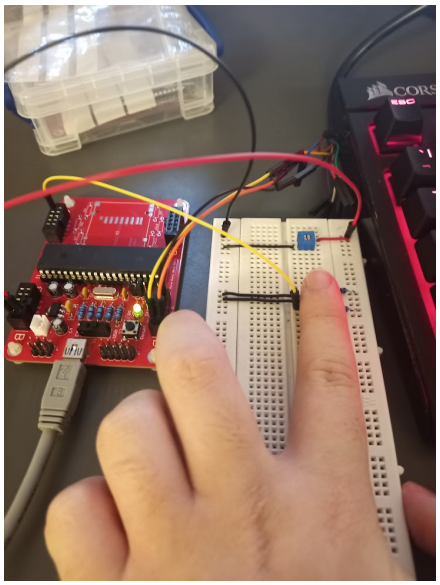
$$noise = log2\ 1024$$
$$noise = 10$$

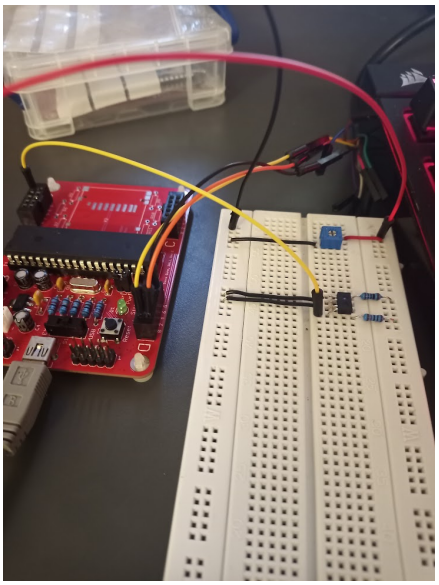
Determining V > Va I am confident that there is nothing in proximity to the sensor, and when V < Vp I am confident that there is an object in proximity to the sensor.

Va = 3.22V n=1000
Vp = 1.61V n=500

I then edited the code so that it would turn the LED on or off depending on the voltage measured on whether the voltage measured at the photosensor.

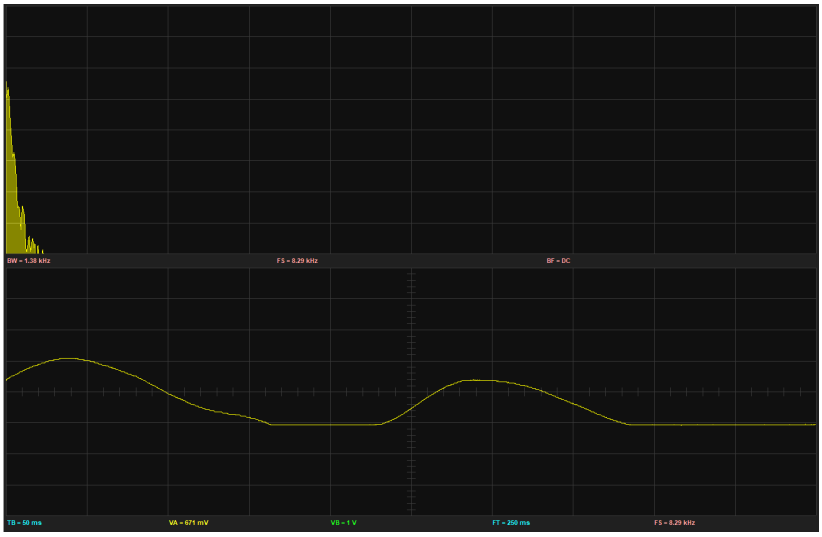
```
void channel_adc(uint16_t n)
{
    if(n < 500)
    {
        PORTB |= _BV(PB7);
        printf("\nLED On\n");
    }
    if(n > 1000)
    {
        PORTB &= ~_BV(PB7);
        printf("\nLED Off\n");
    }
}
```

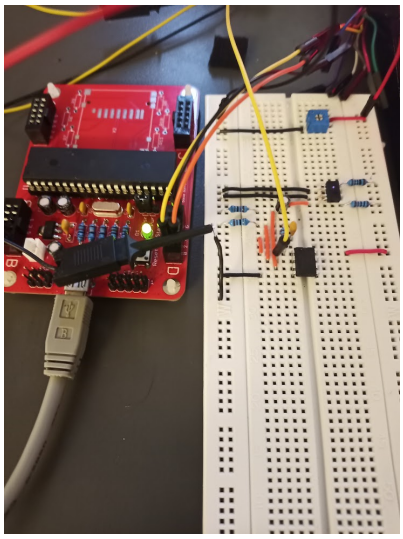




```
LED On
 447 : 1.44194V
 589 : 1.90000V
 824 : 2.65806V
 927 : 2.99032V
 974 : 3.14194V
 995 : 3.20968V

LED Off
1015 : 3.27419V

LED Off
1015 : 3.27419V

LED Off
1016 : 3.27742V

LED Off
1016 : 3.27742V

LED Off
1016 : 3.27742V
```

Using an active band pass filter I was able to get the bitscope to register my pulse using very small fluctuations in the proximity sensor.





I then began modifying the code to calculate my bpm.

```
for (;;)
    {
        result = read_adc(); //get position of an ADC
        if (result > 250)
        {
            if (pulse == 0)
            {
                t_end = t_start;
                t_start = count;
                if (edge)
                {
                    bpm = 60 * 100.0 / (t_end - t_start)
                }
                edge = 1;

                pulse = 1;
            }
            for(;;)
            {
                _delay_ms(10);
               result = read_adc(); //get position of an ADC
               count += 10;
            }
        }
        printf("bpm = %6.5lf\n", bpm);
    }
```

C10

Preparation -

07/12/20

Interrupts

AVR system reset condition - External Pin, Power-on, Brown-out, Watchdog, and JTAG Reset.
Digital Input Interrupts - Interrupts triggered by a signal or a pin change on an input pin.
Timer interrupts - For each counter there are A and B comparator interrupts and an overflow interrupt.
Serial interrupts - Interrupts for serial data transmission being completed.
Analogue interrupts - interrupts for the analogue comparator and for completing a read.

The ‘volatile’ keyword is a declaration for global variables which stops the compiler removing

them in the optimisation stage.

sei(); // Enable interrupts
cli(); // Disable all interrupts

Interrupts commands are in the header:

```
#include <avr/interrupt.h>
```

An example of an interrupt handler in C that acts on the ADC conversion complete event is:

```
    /* Enable Interrupts */
    sei();
    ADCSRA |= _BV(ADIE);
```

An example of an interrupt service routine is:

```
ISR(ADC_vect)
{
    if (ADC > 650)
        PORTB |= _BV(PB7);
    else
        PORTB &= ~_BV(PB7);
}
```

An interrupt reduces the amount of CPU overhead needed to check if the input has been enabled. An excellent example of an interrupt is a temperature sensor in a nuclear power plant. It is unlikely to reach the threshold temperature so checking it repeatedly will take up a lot of processing time but it is critically important that it is detected if it does occur.

09/12/20

I modified the code for the seven segment display so that it would count to 1s using a timer interrupt. This is more accurate than using <util/delay> as it takes the time for a command to be execute.

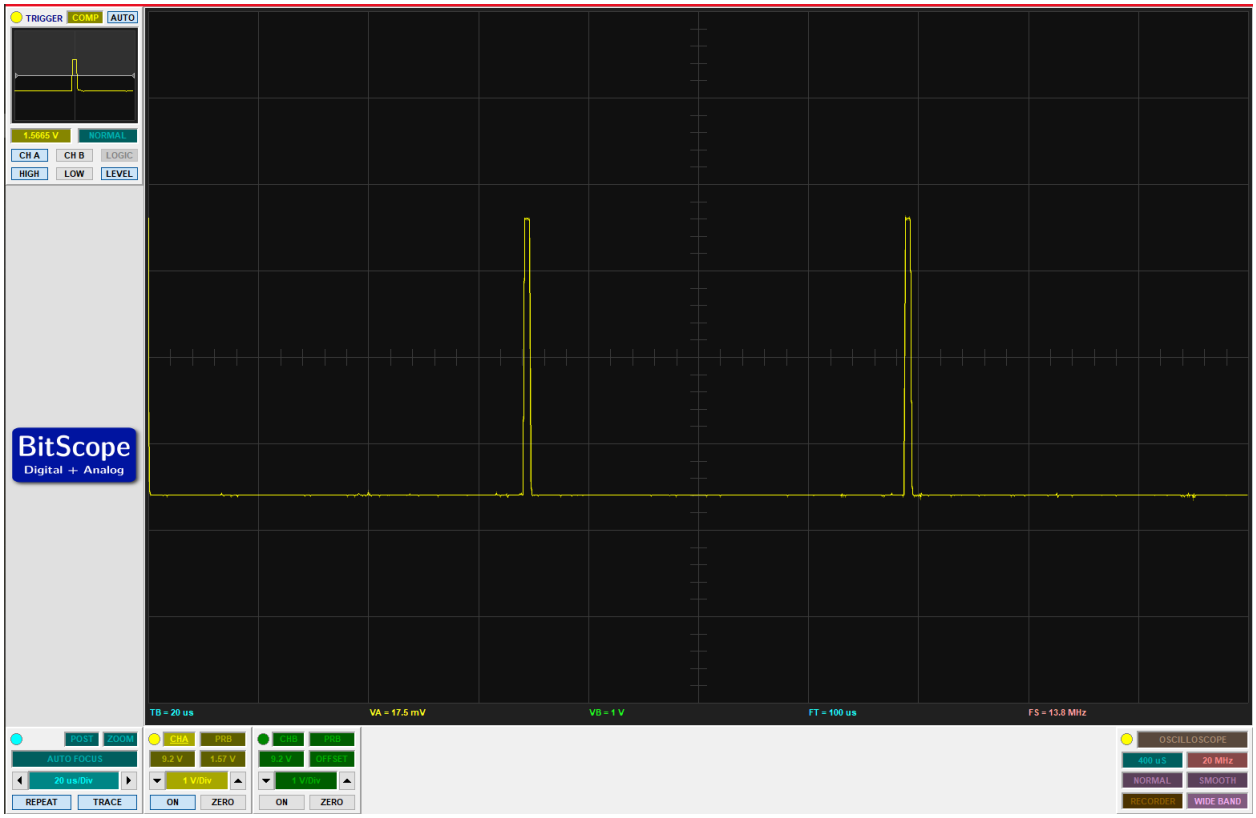
```
const uint8_t segments[10] = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE0,
0xFE, 0xE6};
uint8_t count = 0;

ISR(TIMER1_COMPA_vect)
{
    PORTA = segments[count % 10];
    count++;
}

int main(void)
{
    DDRA = 0xFF; //Sets portA as outputs

    /* Timer 1 CTC Mode 4 */
    TCCR1A = 0;
    TCCR1B = _BV(WGM12);
    /* Prescaler /1024 */
    TCCR1B |= _BV(CS12) | _BV(CS10);
    /* Set timeout at 1000ms */
    OCR1A = 11719;
    /* Enable interrupt */
    TIMSK1 |= _BV(OCIE1A);
    sei();
    for (;;);
}
```

Using the bitscope I was able to verify that my code works. Each segment triggered when it was supposed to.

Using my proximity sensor as a base I created a program that ran in free-running ADC conversion and pulsed an LED on conversion complete.

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

ISR(ADC_vect)
{
    PORTB |= _BV(PB7);
    _delay_us(1); //Delay by 1 microsecond
    PORTB &= ~_BV(PB7);
}

void init_adc(void)
{
    /* Set ADC prescaler to 64 - 187.5kHz ADC clock */
    ADCSRA = _BV(ADPS2) | _BV(ADPS1);
    /* AREF reference, channel 0, right shift result */
    ADMUX = 0x00;
    /* Free-running mode and enable ADC */
    ADCSRA |= _BV(ADATE) | _BV(ADEN);
    ADCSRB = 0x00;
}

int main(void)
{
    init_adc();
    DDRB |= _BV(PB7);
    PORTB &= ~_BV(PB7);

    /* Enable Interrupts */
    sei();
    ADCSRA |= _BV(ADIE);
    /* Start Conversions */
    ADCSRA |= _BV(ADSC);
    for (;;);
}
```

Using the pulses on conversion complete I was able to calculate the maximum sampling rate of the ADC.



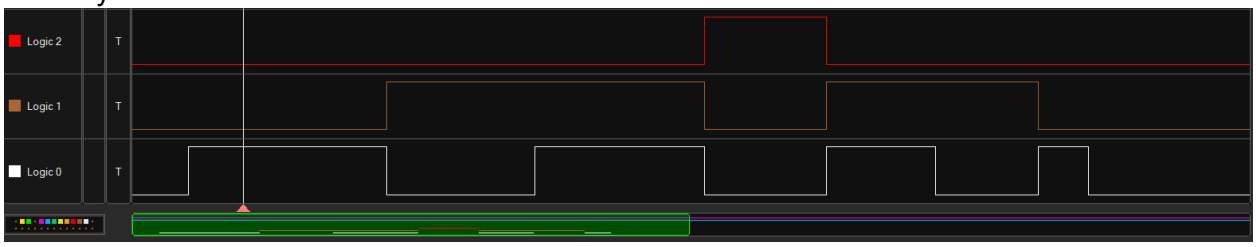
2.9 div * 20 us/div = 58us
f = 1/58E-3 = 17.2 kHz

Using external interrupts I created code that increments and decrements a count when buttons are pressed. Using interrupts meaning the CPU overhead is low.

```
volatile uint8_t count = 0;


ISR(INT0_vect)
{
    if (count > 0)
    {
        count--;
    }

    PORTA = count;
}
```

```
ISR(INT1_vect)
{
    if (count < 255)
    {
        count++;
    }

    PORTA = count;
}

int main()
{
    DDRA = 0xFF;
    DDRB |= _BV(PB7);
    DDRD = 0x00;
    PORTD = 0xFF; //Enables inbuilt pull-up resistors

    /* Set to trigger on falling edge */
    EICRA |= _BV(ISC01) | _BV(ISC11);
    /* Enable interrupt */
    EIMSK |= _BV(INT0) | _BV(INT1);
    sei();

    for (;;);
}
```

Checking with my logic analyser I was able to verify that the count increased and decreased correctly.



Finally I combined all three pieces of code so that I was able to change the sampling rate. My code modified the ADC's sampling rate based on my button presses. However, I was unable to call the ADC ISR I suspect I did not configure the Timer0 ISR correctly.

```
ISR(TIMER0_COMPA_vect)
{
    /* Start Conversions */
    ADCSRA |= _BV(ADSC);
}

ISR(ADC_vect)
{
    PINB |= _BV(7); //Toggle bit n of port x
}

ISR(INT0_vect)
{
    if (OCR0A > 0)
    {
        OCR0A--;
    }

}

ISR(INT1_vect)
{
    if (OCR0A < 255)
    {
        OCR0A++;
    }
}

void init_adc(void);

int main(void)
{
    init_adc();
    DDRA = 0xFF;
    DDRB |= _BV(PB7);
    DDRD = 0x00;
```

```
    PORTD = 0xFF; //Enables inbuilt pull-up resistors

    PORTB &= ~_BV(PB7);

    /* Timer 0 CTC Mode 4 */
    TCCR0A = 0;
    TCCR0B = _BV(WGM12) | _BV(WGM13)  /* varying f with OCR0A */;
    OCR0A = 0xFF;

    /* Set to trigger on falling edge */
    EICRA |= _BV(ISC01) | _BV(ISC11);

    /* Enable Interrupts */
    EIMSK |= _BV(INT0) | _BV(INT1);
    TIMSK0 |= _BV(OCIE0A);
    ADCSRA |= _BV(ADIE);
    sei();

    for (;;)
    {
        PORTA = OCR0A;
    }
}

void init_adc(void)
{
    /* Set ADC prescaler to 64 - 187.5kHz ADC clock */
    ADCSRA = _BV(ADPS2) | _BV(ADPS1);
    /* AREF reference, channel 0, right shift result */
    ADMUX = 0x00;
    /* Free-running mode and enable ADC */
    ADCSRA |= _BV(ADATE) | _BV(ADEN);
    ADCSRB |= _BV(ADTS1) | _BV(ADTS0);
}
```