Lab #3: The Need for Speed

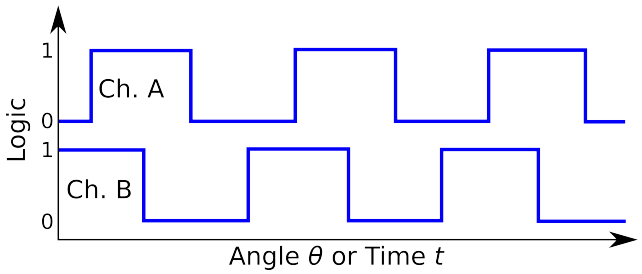
You're going to need at least one more driver for your term project, and learning to write this driver will give you a chance to practice an important programming skill – writing and testing code which runs quickly in response to external events…*quickly.* Your driver will be for an optical encoder; it will measure the rotation of a motor's shaft. You will use a motor with a shaft encoder to test your driver; the same driver will work for most other sorts of encoders you might use in your project as long as you write the driver properly. Because we're also learning to use tasks to implement cooperative multitasking, you will also use tasks derived from the base class `TaskBase` to organize the non-interrupt part of your program. This exercise will be completed by each team in one week.

1 The Project

First make sure everything works. Plug in the motor to a benchtop power supply, set the current limit to one amp, and vary the voltage to run the motor at various speeds. Connect the optical encoder leads, using a second benchtop power supply to supply the encoder with 5 volts DC and setting the current limit at around 50 mA. Most encoders' leads are color coded as follows:

Red or Orange	+5V power
Black or Brown	Ground
Yellow or White	Output A or B
Blue or Green	Output B or A

Use an oscilloscope with two probes to verify that the signal coming from the encoder is a pair of square waves in quadrature. Quadrature means they're about 90° out of phase:



Each time the state (A, B) of the encoder output changes, it is an indication that the encoder has moved one "tick." The particular state change tells the direction in which the encoder has moved: for example, (0, 0) to (0, 1) indicates movement to the right on the graph above, while (0, 1) to (0, 0) indicates movement to the left.

Your task is to write a driver and a test program. The driver will interface with a motor that has an optical encoder, and the test program will exercise the driver thoroughly so that you can exorcise any bugs. You will need to create the following in your program:

- Two interrupt service routines (ISR's), or a single ISR with an alias so that it responds to two external interrupt sources. The ISR's respond to changes in the encoder's outputs by updating a count of where the encoder is (in units of "ticks", meaning how many changes in the state of the A and B channels have been seen). You can use pin change interrupts or external interrupts; the easiest configuration would be to use external interrupts 4, 5, 6, and 7 because these can most easily be configured to interrupt on every transition. Your ISR needs to use `if-else` logic to update the position count every time either channel A or B changes state; this means the ISR must keep track of the previous state of channels A and B and compare that to the current state.

- A driver `class`. An object of this class should set up the needed interrupts and allow the user's code to access the counter which holds the current position. There should also be methods to zero the position and perhaps set the current position to a given value. All transfers of data between the driver class's methods and the ISR **must** be protected from data corruption by shared data items or queues or perhaps critical sections of code.

- A task, implemented as a descendent of class `TaskBase`, which contains code that creates an encoder driver and tests the heck out of it. Your class can print things in the usual way, or even better, it can use `print_ser_queue` to print things (look in `shares.h` and the file with `main()` in it).

The encoder driver should be able to detect if any errors have occurred; in particular, if the encoder's A and B channels change directly from 00 to 11, from 01 to 10, or other changes which indicate missed pulses, the encoder driver must detect this problem and keep some indication of errors. One possible method is to maintain a count of errors, with a method to access that count.

2 Testing

Two properties of the encoder class must be tested: speed and reliability.

- For testing speed, use a variable voltage source to drive the motor at various speeds and an oscilloscope to measure the speeds. The measure of speed that matters is **encoder pulses per second** – the voltage or duty cycle controlling the motor speed is not a useful measure. How fast can encoder pulses be before your encoder class begins to have errors or the program crashes? How does the performance of your

main program change as the motor speed increases? Try having one of your tasks print something out, using a delay loop (not a proper timer) to control printing speed; then see how the speed is affected when thousands of interrupts per second occur.

- For a reliability test, rotate the shaft by hand and compare the encoder's output to an independent method of measuring the angle of rotation. Counting turns (with a fixed angle reference) is OK. Try vibrating the motor or calling it bad names in various languages ("Dumme verfluchte Gerät!" and "ไอ้เครื่องโง่!" have been known to work) and see if the encoder's output remains correct. Encoders are reliable when there are **zero** errors. Even one error is considered a sign of unreliability in an encoder test.

- Try hard to find all possible bugs in your code – you will need a reliable encoder driver for this quarter's term project.

3 Resources

- If you're not familiar with optical encoders, try "Rotary Encoder" on Wikipedia. We're using *incremental,* not absolute, encoders.

- For writing interrupts, you need to know the interrupt sources for the processor and the pins to which they are connected; this information is in the ATmega1281 datasheet.

- The pins which are connected at the edge of the ME405 board are labeled on the edge of the board. For powering the encoders, remember that there are +5V and ground pins on the edge of the board, and these pins make available to you the power from the onboard voltage regulators. The pins you should use are near the middle of the bottom edge of the board; these are the digital 5V and ground.

- You need to look in the `avr-libc` documentation, in particular the documentation for `<interrupt.h>`, to see how interrupts are coded in our AVR environment. Notice the large table in which the interrupt vectors are declared with the `XYZ_vect` names. To find the right ISR name for your processor, you can search within the page for the processor name.

- Some of the files in the `../lib` directory use interrupts. For example, the file `rs232int.cpp` has an interrupt service routine.

4 Formatting Notes

Keeping code neat and readable is important. Here are some tips:

- Set the tab width in the editor to four spaces so that printouts are much neater. Choose the Settings → Configure Kate... menu item. In the configuration dialog box, choose the Editor Component → Editing category and the General index tab. Change the Tab width: setting to 4 characters and click OK.

- When showing the names of functions and variables in our memo, use a fixed width font such as `Courier`. Function names are followed by parentheses, as in `my_func()`.

5 Deliverables

Next week you should turn in a report including the following:

- A one-page memo describing your driver and test program. Test results for speed and reliability should be included; the specific bad words you used to harass the encoder during testing should not. Show the location of your Mercurial repository in the memo.

- A double-sided printout of the Doxygen manual for your encoder driver only. This manual should be complete enough so that someone could successfully use your driver (including connecting the encoder) using the manual as a guide.

- A printout of your task class and test program code. Again, please print double sided.

- You may be asked to demonstrate the operation of your program in lab on the due date.