ENGT5259 EMBEDDED SYSTEMS
Laboratory guidance notes
Part 4: Rotary Encoder

Author: M A Oliver, J A Gow *v1.0*

Table of Contents

Version record

Version	Notes	Date	By
1.0	Initial version	22/03/22	MAO/JAG

PLEASE READ THIS CAREFULLY

This is the fourth of a series of guidance documents to get you started with your embedded systems project. They are aligned with the key aspects of the coursework specification, and thus this document should be read in conjunction with the coursework specification document. These documents will start off with some fairly close guidance - however as you develop your skills the notes will provide less and less detailed guidance and become more of an overview.

Note that the implementation is yours, and **you are encouraged to experiment**, or to try different approaches. As long as (a) you clearly understand the code you have written and (b) your result complies with the specification, you will pass. There is no 'right answers' in

this module. The benchmarks are: (a) have you hit the learning outcomes and (b) does your final project code comply with the specification.

Note that this module is not a C/C++ programming course. Most embedded system development is carried out in C/C++ and you should have had exposure to these languages as part of your undergraduate studies. If you are rusty, or are not familiar with this language there are plenty of tutorials available online (some links are available on the Blackboard shell) that can guide you. It is possible to learn C/C++ alongside this project. They are not especially 'difficult' languages to learn when you are using them to solve a problem and there are some 'templates' available to help.

You are advised to read this worksheet in its entity before commencing, as it contains useful information throughout.

1 Aims and objectives

This document covers an introduction to the implementation of the requirement of section 3 of the coursework specification. This should take you about 1 laboratory session to work through. It also details how the provided template files can be integrated into your existing project.

1.1 Rotary Encoder

This work focusses on the development of firmware for reading and interpreting the rotary encoder on the ATmega328P Microcontroller Development Boards that are located in Q1-01 / Q2-01.

The rotary encoder contains a push switch (which will not be a requirement for this project) and two outputs: Channel A and Channel B. The schematic below shows this subsystem.

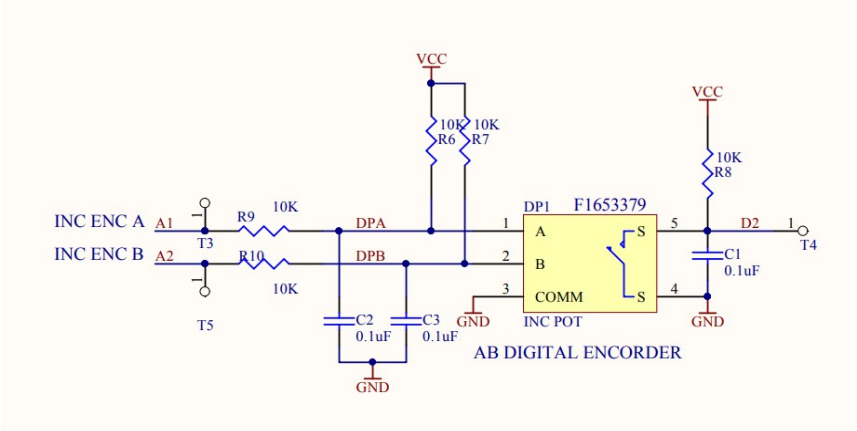


Figure 1: Rotary Encoder Subsystem

From Figure 1, Channel A connects to pin A1 on the Arduino Uno (pin PC1 on the ATmega328P), whilst Channel B connects to pin A2 on the Arduino Uno (pin PC2 on the ATmega328P).

The spindle of the rotary encoder can be turned in discrete steps. Try turning the spindle to get a feel for this. When the spindle is turned the two logic-level outputs of Channel A and Channel B will vary; note that Channel A and Channel B are in quadrature (i.e. 90 degrees

out of phase). A change in state on Channel A causes an interrupt to be generated; we are interested on what happens on a rising edge.

Figure 2 shows how Channel A and Channel B vary when the spindle is turned anticlockwise. Rotating the spindle anticlockwise causes a pulse train to be emitted from Channel A, and a quadrature pulse train to be emitted from Channel B. The rising edge of Channel A raises an interrupt. The corresponding Interrupt Service Routine (ISR) should be written to verify the interrupt and determine the direction of rotation. **If the state of Channel A (PC1) is logic high, *and* the state of Channel B (PC2) is also logic high, this indicates anticlockwise rotation.**

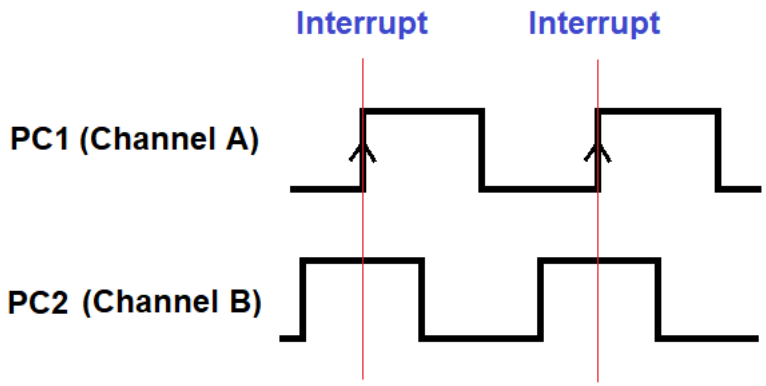


Figure 2. The outputs of the rotary encoder for anticlockwise rotation.

Figure 3 shows how Channel A and Channel B vary when the spindle is turned clockwise. Rotating the spindle clockwise also causes a pulse train to be emitted from Channel A, and a quadrature pulse train to be emitted from Channel B. An interrupt is raised on the rising edge of Channel A. The corresponding Interrupt Service Routine (ISR) should be written to verify the interrupt and determine the direction of rotation. **If the state of Channel A (PC1) is logic high, *and* the state of Channel B (PC2) is logic low, this indicates clockwise rotation.**

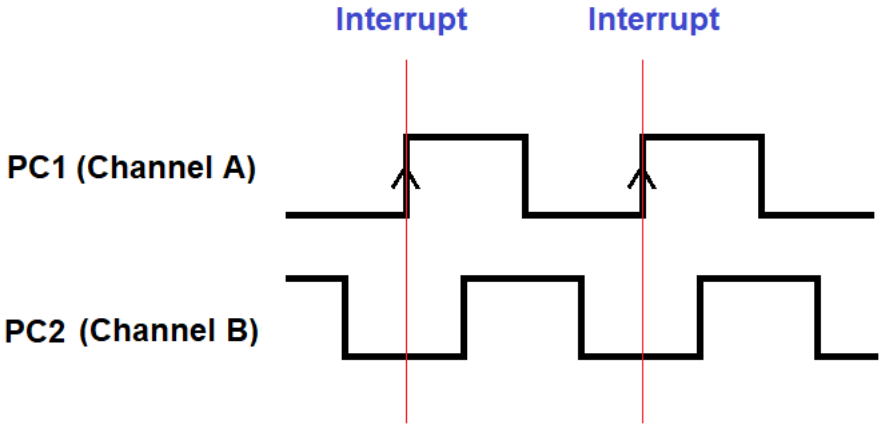


Figure 3. The outputs of the rotary encoder for clockwise rotation.

1.2 Operation

On turning the spindle of the rotary encoder clockwise, the Demand RPS (Revolutions per second) should increase by +1 on every click.

On turning the spindle of the rotary encoder anticlockwise, the Demand RPS should decrease by -1 on every click.

1.3 Encoder Module

This contains two files: encoder.h and encoder.cpp

This is the module for reading the rotary encoder. This is partially complete. It is your task to complete this module.

2 Integrating the template into your current project.

The following steps are useful for integrating the two module files from the template into your current project.

2.1 Adding the files

Before embarking on this part of the process, please ensure that you have closed down your integrated development environment (most likely the Arduino IDE). ***You would be advised to back up your existing work at this stage before proceeding.***

The following files should now be copied into your containing directory (folder):-

encoder.h

encoder.cpp

For those using the Arduino IDE, when you restart the application, these files should now appear in your workspace.

2.2 Creating new message IDs

Examine the code inside the file common.h. It currently contains several message IDs.

For the encoder we require a single new message ID to reflect when the spindle has been turned by 1 click.

Therefore, in common.h, you will need to add a new message ID (with a unique ID number) for the following message:-

- Encoder

You are advised to retain the naming convention that has been used with the existing message IDs.

2.3 Initialization of the encoder module

Identify the function `UserInit()` in your .ino file. You will need to add a single function call

```
ENCInitialize();
```

to initialize the encoder functionality.

Please note that the order in which functions are called within `UserInit()` is important.

Also, remember to `#include` the Encoder module into this file otherwise you will run into compilation errors.

2.4 First compilation

Assuming your code from all previous exercises completely compiles, your updated project containing the template should also compile successfully. ***At this stage you are advised to back-up your work.***

2.5 Control module

In this update, you have skeleton code in control.cpp that can be used to test some of the functionality of the encoder module. This can now be copied into your build. Don't try compiling the code until section 2.6 is complete. Details of this module will be discussed later.

2.6 RPM to RPS

The characteristics of the motor mean that we will have to work with Revolutions Per Second (RPS) rather than Revolutions Per Minute (RPM).

For consistency, you are recommended to change all instances of “RPM” to “RPS”, and all cases of “rpm” to “rps” throughout your code.

Please ensure the code compiles at this stage before progressing.

3 Implementing the encoder functionality

Your next task is to implement the encoder functionality, according to section 3 of the specification. There is a framework already present in the demo code to help get you started.

From the template, the header file for the display module (encoder.h) should not require any modification.

The source file for the display module (encoder.cpp) is essentially a skeleton template that needs development.

3.1 encoder.cpp Preliminaries.

The module contains two functions.

`ENCInitialize()` is used to initialize the encoder module. This is complete.

`ISR(PCINT1_vect)` is an Interrupt Service Routine (ISR) associated with the PCINT1 interrupt vector. You are required to complete this.

3.2 Encoder Module Initialization.

The first line of code in `ENCInitialize()`

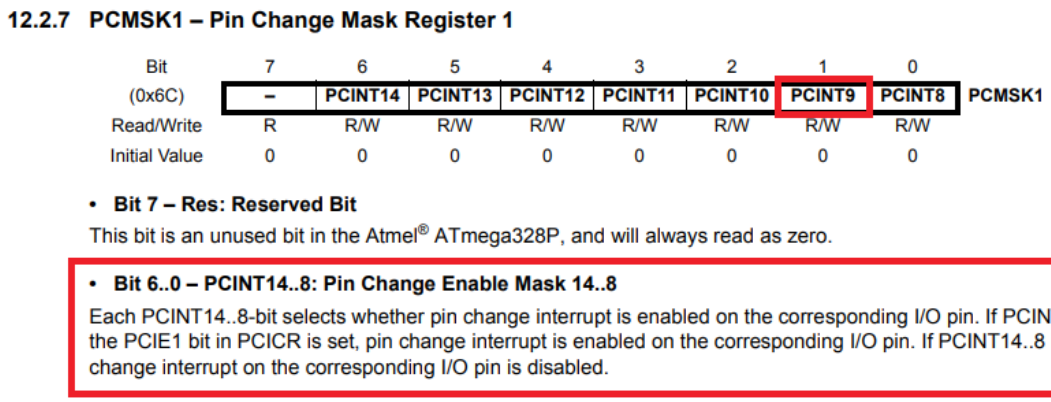
```
DDRC &= ~0b00000110;
```

sets pins PC1 and PC2 on the ATmega328P (A1 and A2 on the Arduino Uno) to inputs. Input pull-up resistors will not be configured in firmware owing to the physical pull-up resistors on the board.

The second line of code

```
PCMSK1 |= 0b00000110;
```

selects pin change inputs on interrupt PCINT9. Please refer to Figure 4. From the ATmega328P datasheet PCINT9 is associated with pin A1.



12.2.7 PCMSK1 – Pin Change Mask Register 1

Bit	7	6	5	4	3	2	1	0	
(0x6C)	–	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8	PCMSK1
Read/Write	R	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – Res: Reserved Bit**

This bit is an unused bit in the Atmel® ATmega328P, and will always read as zero.

- **Bit 6..0 – PCINT14..8: Pin Change Enable Mask 14..8**

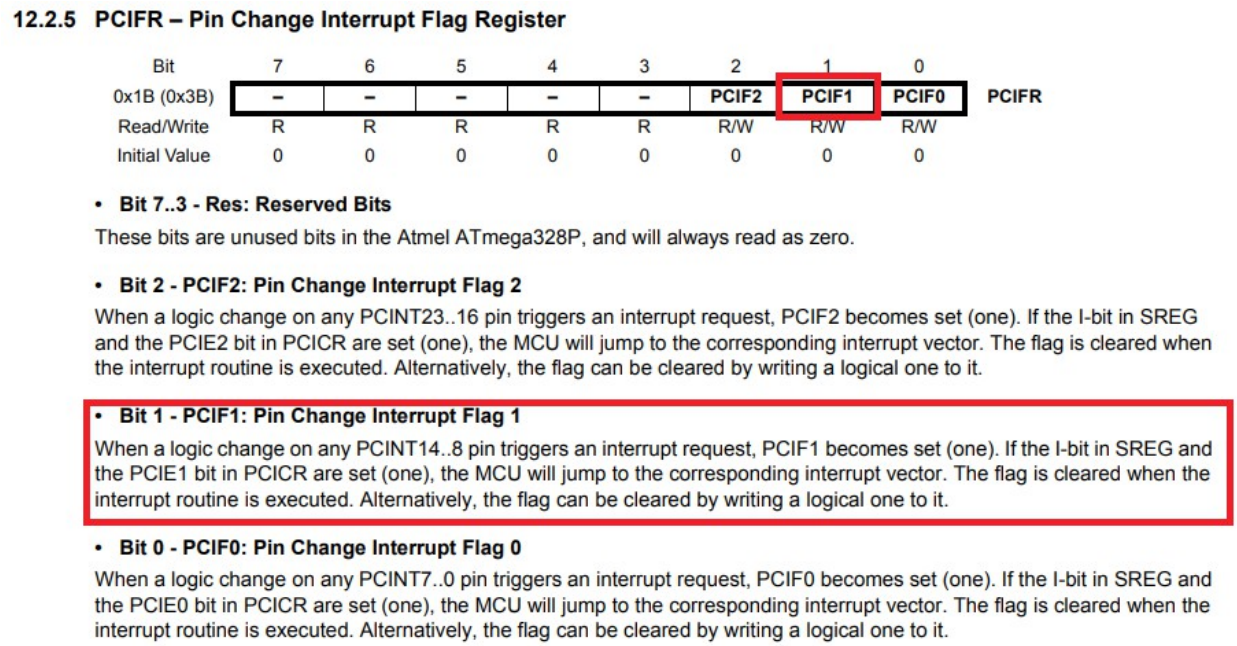
Each PCINT14..8-bit selects whether pin change interrupt is enabled on the corresponding I/O pin. If PCINT14..8 is set and the PCIE1 bit in PCICR is set, pin change interrupt is enabled on the corresponding I/O pin. If PCINT14..8 is cleared, pin change interrupt on the corresponding I/O pin is disabled.

Figure 4. PCMSK1 Functionality (taken from the ATmega328P datasheet by Atmel / Microchip)

The third and final line of code

```
PCICR |= 0b00000110;
```

selects the Pin Change Interrupt Flag 1. An interrupt on PCINT9 causes the Pin Change 1 Interrupt Service Routine to be called. Please refer to Figure 5.



12.2.5 PCIFR – Pin Change Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0	
0x1B (0x3B)	–	–	–	–	–	PCIF2	PCIF1	PCIF0	PCIFR
Read/Write	R	R	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7..3 - Res: Reserved Bits**

These bits are unused bits in the Atmel ATmega328P, and will always read as zero.

- **Bit 2 - PCIF2: Pin Change Interrupt Flag 2**

When a logic change on any PCINT23..16 pin triggers an interrupt request, PCIF2 becomes set (one). If the I-bit in SREG and the PCIE2 bit in PCICR are set (one), the MCU will jump to the corresponding interrupt vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it.

- **Bit 1 - PCIF1: Pin Change Interrupt Flag 1**

When a logic change on any PCINT14..8 pin triggers an interrupt request, PCIF1 becomes set (one). If the I-bit in SREG and the PCIE1 bit in PCICR are set (one), the MCU will jump to the corresponding interrupt vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it.

- **Bit 0 - PCIF0: Pin Change Interrupt Flag 0**

When a logic change on any PCINT7..0 pin triggers an interrupt request, PCIF0 becomes set (one). If the I-bit in SREG and the PCIE0 bit in PCICR are set (one), the MCU will jump to the corresponding interrupt vector. The flag is cleared when the interrupt routine is executed. Alternatively, the flag can be cleared by writing a logical one to it.

Figure 5. PCICR Functionality (taken from the ATmega328P datasheet by Atmel / Microchip)

3.3 Interrupt Service Routine

You are provided with a skeleton interrupt service routine:

```
ISR(PCINT1_vect)
{

}
```

This is mapped to the PCINT1 interrupt vector. The configuration of the code means that this interrupt service routine is invoked on a change of state of Pin PC1 on the microcontroller (A1 on the Arduino Nano). Therefore, a change of state of Channel A of the rotary encoder calls the ISR.

This ISR function is empty. You will need to complete this.

Firstly, you will need to determine the state of Channel A. If this is low, then there is nothing further to do. Otherwise, you will need to determine the state of Channel B.

- If Channel B is low, this indicates clockwise rotation. You will need to POST the value +1 to the message ID associated with the encoder.
- If Channel B is high, this indicates anticlockwise rotation. You will need to POST the value -1 to the message ID associated with the encoder.

3.4 Test Code

The file control.cpp contains some test code used for testing your interrupt service routine. However, this is incomplete and requires further development.

You are provided with a skeleton callback function called `CTRLEncoderClicked()`. This function is subscribed to the Encoder Message ID in the message queue (see line 67). The skeleton code is shown below.

```
void CTRLEncoderClicked(void * context)
{
  int delta = (int)context;


}
```

The value of delta will be either +1 for clockwise rotation, or -1 for anticlockwise rotation.

You will need to add code to this function to:-

- Add the value of `delta` to the variable `demandrps`.
- Constrain the value of `demandrps` to between `RPS_MIN` and `RPS_MAX`.
- Post the value of `demandrps` to the New Demand ID message ID in the message queue.

3.5 The task

You will be provided with skeleton code modules that you will need to add to your project. These contain some sample code that should help you with certain aspects, such as setting-up the interrupt hander.

However, there are regions of the code that you will need to implement. Study the comments. If you see a comment starting “TODO: ”, this will indicate the coding requirements that need to be implemented below that comment.

Once everything is implemented, testing and working, you will have already completed and implemented the majority of Requirement 3 of the specification!