

ELE538 Microprocessor Systems

Lab 6: Wheel Counter Interrupt

Peter Hiscocks
Department of Electrical and Computer Engineering
Ryerson Polytechnic University
phiscock@ee.ryerson.ca
October 22, 2001

Contents

1 Overview	1
2 Wheel Rotation Detectors	2
3 Designing the Rotation Counters	3
4 Software Overview	3
5 Technical Notes on the <i>Input Capture</i> System	5
6 Assignment	6
6.1 Foreground Program	6
6.2 Wheel Counting Routines	7
6.3 Debugging Hints	7
7 Appendix: Interrupts and The Buffalo Monitor	8

List of Figures

1 Wheel Rotation Detectors	2
2 Input Capture System, One Channel	4
3 Wheel Count Interrupt, One Channel	4
4 Wheel Rotation Detectors	5
5 Buffalo Monitor Interrupt Vector Jump Table	9

1 Overview

The *eebot* is equipped with rotation counter circuitry on each of the drive motor outputs. This allows the microprocessor to measure the distance travelled by each of the driving wheels, information that can be used in distance measurement (odometry) or for precise control of wheel rotation to generate accurate turns and accurate straight line motion.

In this exercise the objective is to detect and count the wheel rotation pulses, and display the wheel counts on the Liquid Crystal Display.

The counting system uses the interrupt system of the 68HC11 timer mechanism.

2 Wheel Rotation Detectors

For each motor, one of the gearbox gears has holes in it that are detected by a *photointerrupter*¹. There are 4 holes in the gear and a 13:1 ratio between the gear rotation and the rotation of the output shaft. Counting the holes gives a measurement of distance travelled by each drive wheel, with a resolution of about 5mm distance moved per count².

The count information is connected to computer *input capture* inputs so that the computer can monitor wheel count. It can then be used by the computer to ensure that the robot is moving in a straight line or executing a sharp turn through a desired angle.

The photointerrupters are each connected to a ROTATION LED on the rear deck of the mainframe. When the motors are running at slow speed, these LEDs can be seen flashing.

The simplified circuit for the rotation detector circuits is shown in figure 1.

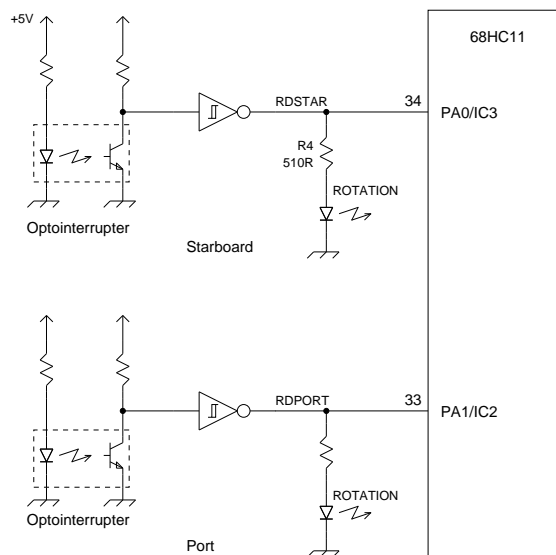


Figure 1: Wheel Rotation Detectors

¹The photointerrupter consists of a light-emitting diode and a phototransistor with a space between them. If something blocks the light path between the two the phototransistor turns OFF. If the optical path is transparent, the phototransistor turns ON. By detecting the state of the phototransistor (ON or OFF), it is possible to sense the passage of a mechanical object through the gap.

²The exact distance travelled depends on the exact wheel diameter, which will change with tire wear and payload. This figure should be considered approximate.

3 Designing the Rotation Counters

Each pulse of the wheel rotation counter corresponds to 5mm or so of movement. How many bits should we allow in the wheel rotation counters? Software counters come in increments of one byte, so the choice is between 8, 16 and possibly 24 bits.

Doing the measurements in meters, an 8 bit counter would provide a maximum count of $(2^8 - 1) \times 0.005 = 1.275\text{m}$, or about 1.2 meters. A 16 bit counter would provide $(2^{16} - 1) \times 0.005 = 327\text{m}$, or about a third of a kilometer. It is difficult to imagine *eebot* travelling more than this distance between program resets, so a 16 bit (2 byte) counter would seem to be sufficient³.

Many of the distances of interest will be associated with robot turns. For example, in a right-angle turn manoeuvre one wheel stops while the other drives and turns the robot through a 90° angle. The rotating wheel will travel through a total distance of 6cm or so. This corresponds to a rotation count of (using units of meters) $0.06/0.005 = 12$. So the usual counts of interest will be quite small and can easily be contained in two bytes.

By simply counting the number of wheel rotation pulses, we are measuring the *absolute* value of the distance travelled. This is usually the information needed by the guidance program. For example, we may want the starboard wheel to drive forward 2cm while the port wheel drives backwards by 2cm. The vehicle will turn sharply to port *in place*, ie, without advancing in distance. The guidance program sets the rotation direction for each wheel and then counts off rotation pulses until desired number have occurred⁴.

4 Software Overview

Now we will examine what machinery is available on the 68HC11 to provide the counting function.

The input capture system of the 68HC11 microprocessor includes three *Input Capture* channels, IC1, IC2 and IC3. Each input capture channel contains circuitry that can automatically *time stamp* an external event so that a computer program can later determine when that event occurred. For example, input capture could be used in a 68HC11 system to measure the width of an electrical pulse. The time stamp is obtained by capturing the count in the free running counter, described in the *Notes for Lab 4* and *Chapter 10, The Pink Book*. A block diagram of one of the three input capture channels is shown in figure 2.

In our case, we only need to detect and count each transition on the input capture pin, so the time-stamping section of the input capture system is not used. The input capture system is used only to generate an interrupt each time a wheel count pulse occurs. This reduces the input capture system to the elements shown in figure 3.

The software to count input transitions is much like the Timer Overflow interrupt routine studied in Lab 4.

Each of the two input capture channels IC3 and IC2 is set up so that a positive transition causes an interrupt. Upon an interrupt, the 68HC11 interrupt hardware transfers machine control from the *Main* program to an interrupt service routine (ISR) which then increments or decrements the wheel rotation counter. The last instruction

³We're assuming that the distance we wish to measure is the distance since the program was last started. If we wanted to keep track of the total distance travelled by the robot (to determine when the engine oil should be changed, for example) then we would have to record the distance count during power off conditions. Every time the robot was shut down, it would copy the distance count to eeprom. Every time it was powered up, it would use the value in eeprom to initialize the distance counters. In this situation, we would require a larger counter. A three byte counter would provide a maximum distance count of 83 kilometers before rolling over.

⁴Unfortunately, because the vehicle tire diameter is not known precisely and there is some slippage of the driving wheels, keeping track of position by measuring direction and distance travelled (known as *dead reckoning* or *odometry*) results in errors that grow with time and soon become too large to be useful. Odometry is a useful navigational tool only over small distances.

To navigate by odometry, you would keep track of the direction as well as the distance travelled by each wheel. Then the starting point would be location zero and distances would be measured positive or negative from this origin. For example, when the port wheel drives forward by 2cm and the starboard wheel backward by 2cm the vehicle heading has changed by approximately $+90^\circ$ and the net translation of the vehicle is zero. The odometry navigation routine needs to translate wheel rotations into heading and translated distance.

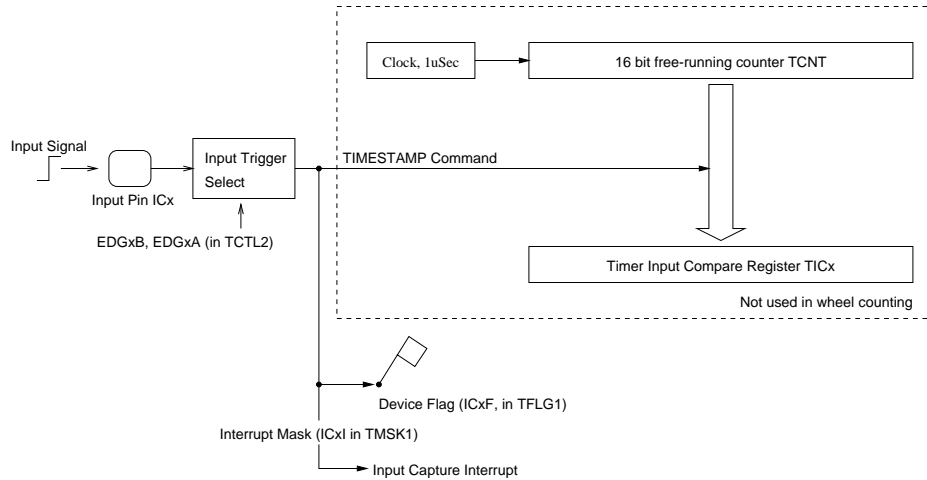


Figure 2: Input Capture System, One Channel

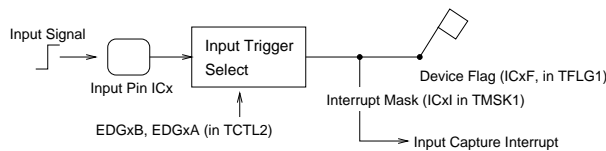


Figure 3: Wheel Count Interrupt, One Channel

in the interrupt service routine is *RTI (Return From Interrupt)*, which returns control back to the *MAIN* program at the point of interruption.

One interrupt service routine could be made to handle interrupts from both the input capture inputs. However, it doesn't require an excessive amount of memory to have a separate *ISR* for each input channel and it simplifies the design, so we'll take that approach. Consequently, the two channels are duplicates: there are two separate interrupt service routines, one for each rotation counter.

Each input channel includes the following features:

- Specification of the *edge polarity*, the polarity of the pulse edge that causes the interrupt.
- A *device flag*, that is set when an input edge occurs and cleared by the interrupt service routine.
- An *interrupt enable flag*, that enables and disables interrupts for this input capture channel.
- An *interrupt vector*, a two-byte address at the top of memory which points to a pseudo-interrupt vector.
- A *pseudo-interrupt vector*, which is a 3 byte *JMP* instruction sending control to the interrupt service routine.
- The *interrupt service routine* for that channel.
- An *interrupt enable* subroutine to initialize the channel machinery and turn on its interrupts.

- A *interrupt disable* subroutine to turn off the interrupts on that channel.

In section 5 below, we discuss each of these components in detail.

Assume that there is a program *Main* which is running in the foreground of the computer. It is directing the operation of the robot and updating the displays. A block diagram representing this system is shown in figure 4.

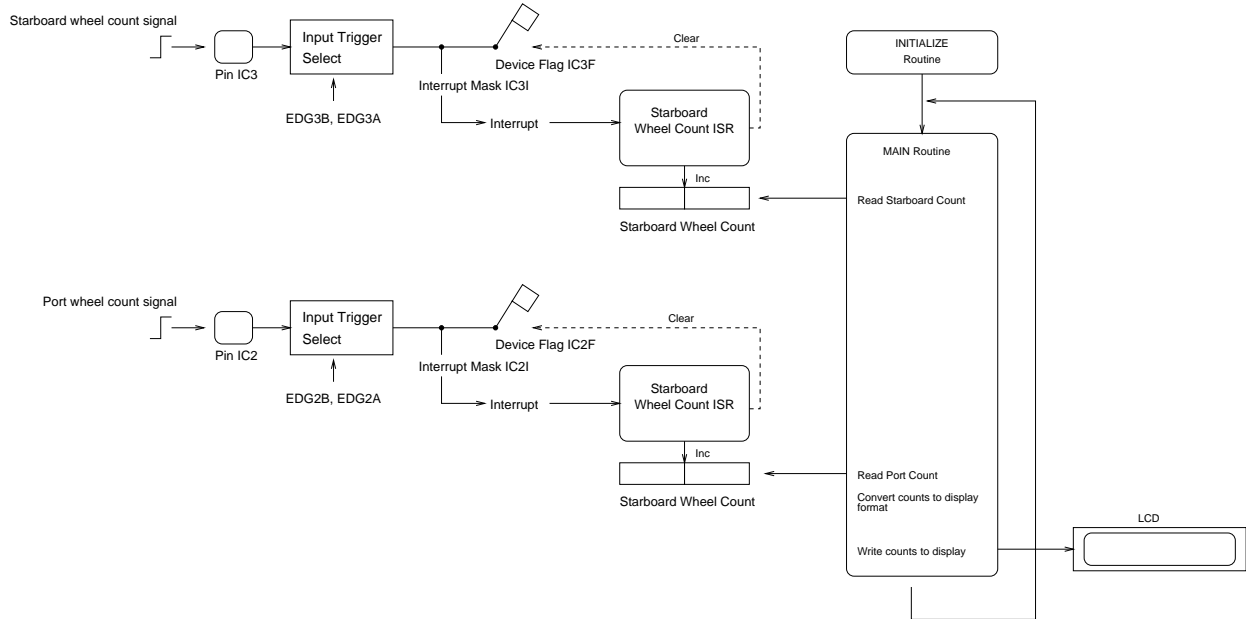


Figure 4: Wheel Rotation Detectors

Notice how the interrupt service routines are called by the input capture interrupts. The ISR each increments its respective wheel count and clears its interrupt flag.

The MAIN routine loops around reading the STARBOARD and PORT wheel count memory locations, converting them from 16-bit binary to a decimal ASCII string, and then displaying the results on the LCD.

5 Technical Notes on the *Input Capture System*

Now the details on each of the features of the interrupt capture machinery (for each channel of the two needed for this lab):

Input Capture Edge Control Two bits select which type of input transitions the capture system will respond to:

- Capture Disabled
- Capture on Rising Edge Only
- Capture on Falling Edge Only
- Capture on Either Edge (Rising or Falling)

In this application, we could select *Capture on Rising Edge* or *Capture on Falling Edge*: in doesn't make much difference. We'll select *Capture on Rising Edge*. See register TCTL2.

Input Capture Status Flag ICxF The ICxF status flag is set to a one each time the selected edge is detected at the input pin. This is the *device flag* for the input capture channel. It must be cleared as part of the interrupt service routine. Remember, timer device flags are cleared by writing a ONE to the flag bit. See register TFLG1.

Input Capture Interrupt Enables When this bit is a zero, the Input Capture Interrupt is disabled. When it is a 1, the interrupt is enabled. We will enable the interrupt, so that each time the device flag is set, an interrupt occurs and the machine jumps to the interrupt service routine. See register TMSK1.

Input Capture Interrupt Vector This is a two-byte address that would normally point to the interrupt service routine. It's located at \$FFEA for interrupt IC3 and \$FFEC for interrupt IC2. (See figure 5.3 of The Pink Book.) However, because the monitor program is present, the reset vector points to the corresponding pseudo-vector, which is located in RAM and must be set up by the interrupt initialization instructions. (See section 7 for more detail on pseudo-interrupts in general.) The pseudo-vector is a jump instruction that directs control to the interrupt service routine.

If we eliminated the Buffalo Monitor and installed our program into EPROM to be executed directly on reset, we would modify the program so that the interrupt vector pointed directly at the interrupt service routine and a pseudo-interrupt vector would not be necessary.

Your interrupt initialization routine does not have to deal with the Input Capture Interrupt Vector – it deals with the Pseudo-Vector described next – but you should be aware of it's purpose and function.

Input Capture Interrupt Pseudo-Vector This is a three-byte location that contains a JMP ISR_ROUTINE instruction, where ISR_ROUTINE is the two-byte address of the Interrupt Service Routine. See figure 5 on page 9. The routine to initialize an interrupt must put the pseudo-vector in this table.

Interrupt Service Routine In this application, the function of the ISR is simply to clear the device flag and then increment (or decrement, depending on the wheel rotation direction) a wheel rotation counter. So the ISR must check the shadow register that contains direction information for its motor, and make an increment or decrement decision based on that bit.

6 Assignment

You are to build a program that demonstrates the wheel rotation counters on *eebot*. This includes both a *Foreground Program* and two interrupt-driven *Wheel Counting Routines*.

6.1 Foreground Program

The foreground routine initializes the interrupt machinery for input capture interrupts IC3 and IC2, and any other variables such as the rotation counters and motor control registers. When all this is in place, it enables the global interrupt signal using the CLI instruction.

Then the MAIN routine enters the Main Loop where it reads the two, two-byte rotation counters, and displays the STARBOARD and PORT rotation counts on the LCDisplay. The display routine should include a software delay so that the display is not written too frequently.

The rotation count should be in decimal format and leading zeros may be shown. The starboard rotation count should be on the top line of the display and the port rotation count on the bottom line.

The initialization routine should start both motors, and we should then see the rotation counts increasing.

Do not attempt to slow either motor by physically touching the *eebot* wheels or gearbox gears. The gearbox has considerable torque and trying to slow down a motor will break something expensive.

The foreground program will call subroutines that

- Convert a hexadecimal number to binary coded decimal
- Convert a BCD number to a printable ASCII string
- Write a string to the LCD
- Position the LCD cursor at the start of the first or second line

These subroutines were provided or developed in previous exercises and should be in your library directory.

6.2 Wheel Counting Routines

You will need two complete interrupt driven rotation counters. The good news is that these routines are almost identical: in most cases, they refer to different bits in the same registers. The wheel counting routines will include

- initialization routine (routine to enable interrupt)
- interrupt service routine
- routine to disable interrupt.

The *routine to disable interrupt* will only be used for debugging purposes.

6.3 Debugging Hints

The following sequence of events may help you debug the routines. We'll assume you're working on the PORT rotation counter.

Check the Initialization

1. Carefully and systematically make sure every timer system register bit that needs to be set or cleared is initialized correctly by this routine.
2. Use the monitor CALL instruction to run the interrupt initialization routine.
3. Check that initialization routine correctly installs the the pseudo-vector 3 byte jump instruction at location \$00E2 or \$00E5. (see figure 5).
4. You won't necessarily be able to check that the timer register bits are set correctly by the interrupt initialization software because returning to the monitor may reset some of the bits.

Check the ISR

1. Write the interrupt service routine as a subroutine (ie, ends with an RTS instruction) and verify that it executes without crashing the machine. Check that every time it is CALLED from the monitor it increments or decrements the rotation counter register correctly based on the bits in the motor direction shadow register.
2. Set the LSByte of the rotation counter to FF and the MSByte to 00. Call the subroutine again. The LSbyte should be 00 and the MSByte 01. Check that overflow of the LSByte causes the MSByte to be incremented correctly.
3. Check that the routine clears the Input Capture flag bit. When all this works correctly change the final RTS instruction to an RTI instruction in preparation for its use as an interrupt service routine.

7 Appendix: Interrupts and The Buffalo Monitor

When we attempt to use the HC11 interrupt vectors under the Buffalo monitor, we are immediately presented with a problem.

- The 16 bit *reset vector* must live in some constant location where the hardware can find it every time the computer resets. In the case of the 68HC11 that location is at the top of memory, in location \$FFFE, \$FFFF.
- The reset vector must reside in permanent memory because it *must* always be there when the processor restarts. Permanent memory, on the MPP board, means EPROM. So the system EPROM must reside at the top of memory to contain the reset vector.
- Each HC11 interrupt vector must each point to an interrupt service routine.
- Like the reset vector, the interrupt vectors reside in fixed locations at the top of memory and thus cannot be changed during the development process without reburning the EPROM.
- During the development process, the locations of the interrupt subroutines will change.

We have a problem because the interrupt vectors must be contained in EPROM but must also be able to change so that they always point at the corresponding interrupt service routine. The solution used by the Buffalo monitor to reconcile these two requirements is to have the interrupt vectors point to a block of RAM memory, the *Interrupt Vector Jump Table*. The Interrupt Jump Table resides in the 68HC11 internal RAM between locations \$00C4 and \$00FF. Each interrupt vector points to a three byte *jump absolute instruction* in this table. Thus, on an interrupt, the processor reads the corresponding interrupt vector (in EPROM, at the top of memory). This sends the processor to the corresponding jump instruction in the jump table. The jump instruction then sends the processor to the interrupt service routine.

Because these jump instructions (called *pseudo-vectors*) are in RAM rather than EPROM, they can be easily changed to point to the start of an interrupt service routine, wherever it lives in memory.

There are three different aspects of this to keep in mind:

- If you are using an interrupt, your program must put a three byte absolute jump instruction in the appropriate Interrupt Vector Jump Table entry, to point at the corresponding interrupt service routine.
- It now takes three more machine cycles to execute the jump instruction and get to the interrupt service routine, not usually a major consideration.

- The Interrupt Vector Jump Table takes up some 60 bytes of the processor internal RAM. This could be a problem for an application trying to use internal RAM without adding external RAM. Naturally, Ugly Things Happen if your program writes over this table and you are using interrupts.

After reset, the Buffalo monitor initializes the 20 entries of the Interrupt Vector Jump Table to point to a monitor routine called `STOPIT`. If an interrupt occurs and the entry in the Interrupt Vector Jump Table has not been changed from its initial value, `STOPIT` brings the program to a halt. This is preferable to having the program branch to some random address or execute some random instruction.

The RAM addresses for the Interrupt Vector Jump Table are shown in figure 5.

Interrupt Function	Jump Table Address
Clock Monitor	\$00FD-00FF
Computer Operating Properly	\$00FA-00FC
Illegal Opcode	\$00F7-00F9
Software Interrupt (SWI)	\$00F4-00F6
XIRQ (External Interrupt Request)	\$00F1-00F3
IRQ (Interrupt Request)	\$00EE-00F0
Real Time Interrupt	\$00EB-00ED
Timer Input Compare 1	\$00E8-00EA
Timer Input Compare 2	\$00E5-00E7
Timer Input Compare 3	\$00E2-00E4
Timer Output Compare 1	\$00DF-00E1
Timer Output Compare 2	\$00DC-00DE
Timer Output Compare 3	\$00D9-00DB
Timer Output Compare 4	\$00D6-00D8
Timer Output Compare 5	\$00D3-00D5
Timer Overflow	\$00D0-00D2
Pulse Accumulator Overflow	\$00CD-00CF
Pulse Accumulator Input Edge	\$00CA-00CC
Serial Peripheral Interface (SPI)	\$00C7-00C9
Serial Communications Interface (SCI)	\$00C4-00C6

Figure 5: Buffalo Monitor Interrupt Vector Jump Table

For example, if you installed an 'illegal opcode' interrupt handler at \$49A3, you would install the `JMP` opcode \$7E at location \$00F7 in the table and the word \$49A3 at location \$00F8.

Notice that the Software Interrupt is used by the Buffalo Monitor as part of the breakpoint mechanism, so is not available for any other use.