Patrick Austin

CPE 301 - 1104, Fall 2016

Homework 8

11/15/2016

1. An interrupt is used to notify the CPU of an event that has occurred so that the system can respond to the situation without the need to spend the processor’s time and resources continuously polling an input. To put it another way, interrupts seem to usually be a way to send data relating to external hardware (a timer, an ADC, the UART, some other external device via GPIO, among others) to the main program without using polling, which is costly form a performance standpoint.

2. When an interrupt occurs, the CPU finishes processing its current instruction and then temporarily halts processing main program instructions from the program counter. The context of the CPU is saved to the stack. The code associated with the interrupt, the ISR, is then executed instead. At the end of the ISR the CPU context is restored to its original state, and the program counter resumes executing instructions from the next location in the program counter, where it left off when the ISR interrupted it.

3. The Atmega328P processor accepts interrupts from, among others: the system reset button; external pins via GPIO; the timers via various means such as on counter overflow or when the counter matches a certain compare value; the UART when eg transmit or receive is complete or when the data register is empty; and the ADC when a conversion is complete. Consult table 9.1 in the textbook for the complete list.

From a compiler point of view, the Atmega328P AVR-GCC compiler supports interrupts via a system of predefined macros that provide the programmer resources to handle, among other things: the priority of interrupts; the degree of compiler-generated CPU context management code; and the permissibility of interrupting an interrupt.

4. Interrupt priority refers to the order in which interrupts will be executed if multiple interrupts occur while some other interrupt is being handled by its ISR. On the atmega this priority is predefined in the order of priority shown in table 9.1 in the textbook, with the priority 1 interrupt (reset) trumping all the others.

Normally interrupts are disabled while an interrupt is being processed, so the current ISR will finish before the next one will be selected (in order of priority) and executed. It is, however, possible to use the ISR\_NOBLOCK ISR to allow nested interrupts, where an interrupt can itself be interrupted by an interrupt with higher priority. This is made possible by enabling interrupts even during the processing of the ISR\_NOBLOCK.

5. To me this question seems to depend on what one means by ‘system designer’.

If we are talking about the original designers of the board/environment/system, they had to create a system of priority of interrupts and design the hardware that would allow the CPU to ‘listen’ for interrupts between instructions, interject ISR code, and then resume normal operation from the program counter. They also had to design the external systems such as the timer, the ADC, the UART, to be able to send interrupts in a way that would link up correctly with the CPU. The designers of the compiler had to generate eg the system of macros that allows a programmer to create ISRs that work correctly with the hardware.

If we are talking about a designer programming for some board/environment/system (I use the Arduino board/environment as an example), properly configuring an interrupt is a question of correct programming and perhaps circuit design depending on the project in question.

To properly configure an interrupt using the timer, for example, one must make sure global interrupts are enabled in SREG, initialize the timer such that the desired interrupt is enabled and will be sent in the situation the user desires (ie at some desired time interval), create the ISR that will convey necessary information to the main program (usually via global variable), and design a main program that works around the interrupt by avoiding pitfalls such as accessing the global variable as the ISR might write to it, which has the potential to create a garbage value. To accommodate this issue, one disables interrupts while reading the value the ISR might write to.

This process is largely similar for other hardware that might raise an interrupt, such as the ADC, the UART, or a GPIO pin. One initializes appropriately, enables/disables global interrupts as necessary, and creates an appropriate ISR to accomplish the desired objective without any bugs. How’s that for a broad answer to a vague question?

6. Again, trying to accommodate for vagueness: on the atmega one can turn interrupts on or off globally via bit 7 of SREG. This is used to avoid bugs like the one described in the previous problem, or to allow/disallow the interruption of an interrupt as discussed in problem 4. On the level of the individual devices, enabling/disabling the system of interrupts is a question of how the device is initialized. One can initialize a timer on the atmega with any of several interrupts, or none at all, depending on how the registers governing its initialization are configured. See the next problem for an example.

7. Here is the specified code:

//Patrick Austin

//CPE 301 HW 8 Problem 7

//Revision Number 1

//Revision date: 11/14/2016

//hardware pointer declarations

//output

volatile unsigned char\* myPortDDRB = (unsigned char\*) 0x24;

volatile unsigned char\* myPortB = (unsigned char\*) 0x25;

volatile unsigned char\* myPortDDRK = (unsigned char\*) 0x107;

volatile unsigned char\* myPortK = (unsigned char\*) 0x108;

//timer

volatile unsigned char\* myTIFR1 = (unsigned char\*) 0x36;

volatile unsigned char\* myTCC1A = (unsigned char\*) 0x80;

volatile unsigned char\* myTCC1B = (unsigned char\*) 0x81;

volatile unsigned char\* myTCC1C = (unsigned char\*) 0x82;

volatile unsigned int\* myTCNT1 = (unsigned int\*) 0x84;

volatile unsigned char\* myTIMSK1 = (unsigned char\*) 0x6F;

//interrupts

volatile unsigned char\* mySREG = 0x5F;

//global vars for ISR

unsigned int timerCount = 63582; //timer preload value to interrupt in 1/8 second

//using 1024 prescaler. 1/8s = 125 ms

//preload value = 65536 - 15.625 \* 125 = 63582

unsigned char timerMode = 0x05; //TNCT1B value corresponding to 1024 prescaler

ISR(TIMER1\_OVF\_vect)

{

\*myTCC1B = 0; //turn off the timer

\*myPortB = \*myPortB ^ 0x80; //toggle the LED

\*myTCNT1 = timerCount; //reset the preload value

\*myTCC1B = timerMode; //enable timer with 1024 prescaler

}

void setup()

{

\*myPortDDRB = \*myPortDDRB | 0x80; //enable LED output

\*myTCC1A = 0; //enable timer 1 with TOV1 interrupt

\*myTCC1B = 0;

\*myTCC1C = 0;

\*myTIMSK1 = 0x01;

\*mySREG = \*mySREG | 0x80; //enable global interrupts

\*myTCNT1 = timerCount; //get preload value

\*myTCC1B = timerMode; //enable timer with 1024 prescaler

}

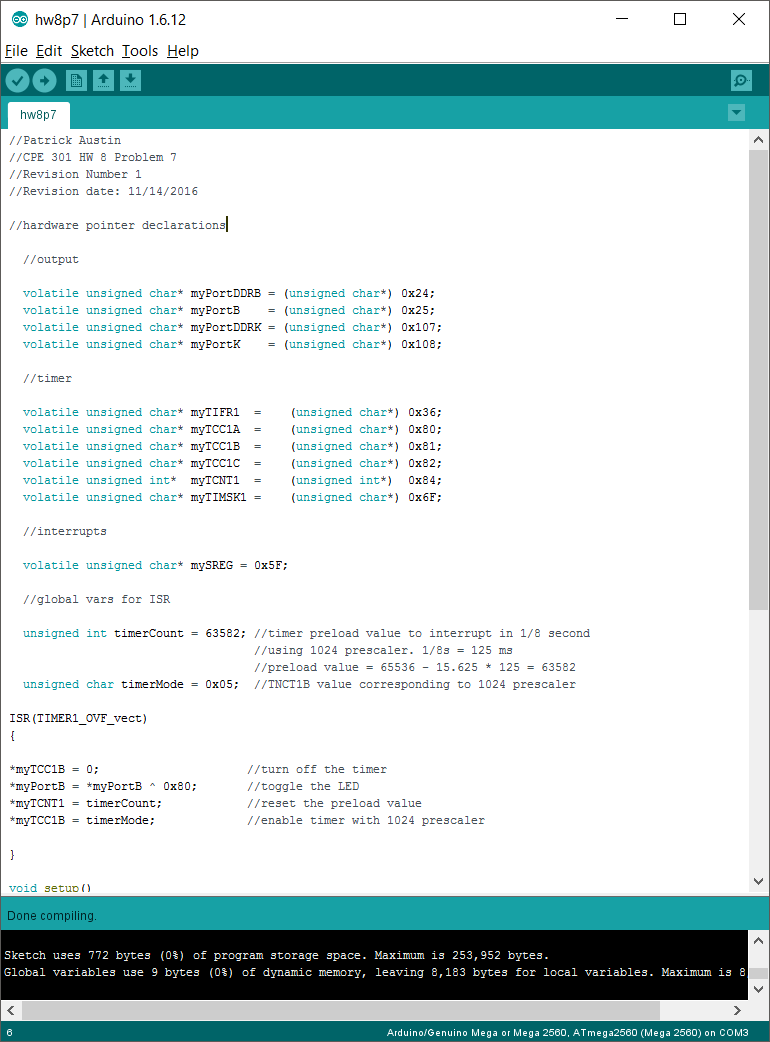
void loop()

{

//wait for the interrupt...

}

Here is an image showing the program compiling:



8. Here is the specified code:

//Patrick Austin

//CPE 301 HW 8 Problem 8

//Revision Number 1

//Revision date: 11/14/2016

//hardware pointer declarations

//output

volatile unsigned char\* myPortDDRB = (unsigned char\*) 0x24;

volatile unsigned char\* myPortB = (unsigned char\*) 0x25;

volatile unsigned char\* myPortDDRK = (unsigned char\*) 0x107;

volatile unsigned char\* myPortK = (unsigned char\*) 0x108;

//timer

volatile unsigned char\* myTIFR1 = (unsigned char\*) 0x36;

volatile unsigned char\* myTCC1A = (unsigned char\*) 0x80;

volatile unsigned char\* myTCC1B = (unsigned char\*) 0x81;

volatile unsigned char\* myTCC1C = (unsigned char\*) 0x82;

volatile unsigned int\* myTCNT1 = (unsigned int\*) 0x84;

volatile unsigned char\* myTIMSK1 = (unsigned char\*) 0x6F;

//interrupts

volatile unsigned char\* mySREG = 0x5F;

//global vars for ISR

unsigned int timerCount = 65376; //timer preload value to interrupt in 1/100000

//second using 1 prescaler. 1/100k s = 10 microsec

//preload value = 65536 - 10/.0625 = 65376

unsigned char timerMode = 0x01; //TCC1B value corresponding to 1 prescaler

ISR(TIMER1\_OVF\_vect)

{

\*myTCC1B = 0; //turn off the timer

\*myPortB = \*myPortB ^ 0x80; //toggle the LED

\*myTCNT1 = timerCount; //reset the preload value

\*myTCC1B = timerMode; //enable timer with 1 prescaler

}

void setup()

{

\*myPortDDRB = \*myPortDDRB | 0x80; //enable LED output

\*myTCC1A = 0; //enable timer 1 with TOV1 interrupt

\*myTCC1B = 0;

\*myTCC1C = 0;

\*myTIMSK1 = 0x01;

\*mySREG = \*mySREG | 0x80; //enable global interrupts

\*myTCNT1 = timerCount; //get preload value

\*myTCC1B = timerMode; //enable timer with 1 prescaler

}

void loop()

{

//wait for the interrupt...

}

Here is an image showing the program compiling:



Since the expected signal period is 20 microsec, the expected signal frequency is 1/20 microsec = 50kHz. The expected pulse width is 10 microsec. Compare this with this result from running the program with the lab oscilloscope:



It’s hard to read in the photo, but the measured frequency is about 40kHz and the measured pulse width is about 12.5 microsec. Unless there is some user error here (can’t count it out!) the microcontroller’s accuracy appears to be not great (~20% error) at this high of a frequency. One could probably use trial and error to find a preload value that actually corresponds to 50kHz, but it is not the value one would expect from the timer calculations we have been accustomed to using in this class.