Low Power and Embedded Systems - Workbook 4

Introduction

In this workbook we will accurately time the interval between two input events present at the microcontroller, investigate the sleep mode of the microcontroller and look at other techniques to reduce power consumption.

Supporting material

atmega168P.pdf	http://www.cl.cam.ac.uk/teaching/0910/P31/docs/atmega168P.pdf Data sheet for the Atmel ATMEGA168P used in these exercises. You will need to refer to this frequently. Within these workbooks this will be referred to as 'the datasheet'. The section numbers referred to in these workbooks refer to revision 8161C of the datasheet dated 05/09.
avr-libc documentation	http://www.nongnu.org/avr-libc/ Documentation for the libraries available on the PCs

An on-line version of this guide is available at:

workbook4.html [http://www.cl.cam.ac.uk/teaching/0910/P31/workbook4.html]

Exercise 1 - Measuring the interval between events

This exercise builds upon the work undertaken in Workbook3, where you connected an LCD to the microcontroller. The objective in this task is to accurately time the interval between two switch change events. In this exercise the switches used are reed switches, which close in the presence of a magnetic field. The microcontroller can be programmed to generate interrupts when inputs change from low to high, high to low, or at either transition. Using two such interrupts and a timer adjusted to count in multiples of 1us it is possible to make an accurate event interval timer, and display the result on the LCD. To get timer ticks at exacly 1us intervals, the device will be run from an 8MHz master clock, and using a timer prescale value P of 8 (note just the timer presale, not the master clock divide which is set in the fuses). The 16-bit timer takes a maximum time of 65ms before it overflows. However we will use a count match interrupt to limit the counter to a maximum period of 10ms, and by counting these interrupts using a 16-bit word this will allow timing of intervals up to approximately 650 seconds to be performed.

1. Copy the code from worksheet3 exercise2.c to form worksheet4 exercise1.c

   Also copy the relevant Makefile, and change exercise2 to exercise1 in 4 places.

   Read through this section to get an overview of the steps before continuing.

2. Change the crystal for an 8MHz one. Change the definition of F_CPU. It is likely that during testing the _delay_ms() and _delay_us() functions will be useful. Also change the baud rate divider in USART_init(), by referring to table 19.11 in the datasheet. Issue a `make clean` command to force a recompile of serial.c now that F_CPU has changed.

3. Fit two reed switches to the prototyping board where they are easily accessible to a passing magnet. One end will connect to 0V, the other ends will be wired to inputs PD2 (pin4) and PD3 (pin5). The pin spacing of the reed switch is not a multiple of 0.1 inch, so you will need to fit the switch at an angle. The magnets are small, so stick the magnet on the end of a screwdriver.

4. Refer to section 13 of the data sheet, and in the port_direction_initialisation part of your code, set PD2 and PD3 to be inputs and enable pullup resistors on these two input pins.

5. Write a simple main() function to read from `PIND` and display the result on the LCD. Use this to check that the switches work as expected, that is the input is low when the switch closes (magnet near) and high when the switch opens (magnet far).

6. With the inputs working, refer to section 12 of the datasheet, and choose a suitable value for register `EICRA` so that a high to low transition generated by a switch closing on PD2 or PD3 triggers an interrupt from INT0 or INT1 respectively. Note that setting `EICRA` merely selects which edge would generate an interrupt. You must also write a value into register `EIMSK` to enable the interrupt.

7. Add code to process these interrupts: ISR(INT0_vect) and ISR(INT1_vect). Initially just do something like display a character on the LCD when a transition is detected.

8. Set up the 16-bit timer1 in CTC mode to count up to 9999 then reset to 0. You might want to refer back to workbook3 exercise 1.

9. Next update ISR(INT0_vect) to clear and start timer1, and ISR(INT1_vect) to stop it. At this stage the timer will overflow, but that will be addressed next.

10. Display the interval on the LCD, using the LCD_Display_* functions from serial_lcd.h which we used in workbook 3, exercise 2. Because the results are in microseconds, the values will look random, but should change each time the switches are triggered in sequence, and should not exceed 9999.

11. You will see that there is a problem with switch bounce. In each pin change interrupt handler, disable the pin change input for the interrupt which has just happened, and re-enable it in the other pin change interrupt. Even this may not happen quickly enough to prevent a second interrupt. Adding 47nF capacitors across the reed switches may be necessary to slow the signal change.

12. To address the overflow issue, create a 16-bit variable that is cleared when the timer is started. Add a Counter Match Interrupt handler to increment this 16-bit value each time the timer count reaches 9999. When the second switch event occurs, this 16-bit value represents a count of 10 millisecond intervals since the timer started. Write this 16 bit value to the LCD followed by the timer1 value, go get a microsecond count. You will need to create a modified version of LCD_Displayint16 to output the value of timer1 as 4 characters including any leading zeroes (the existing LCD_Displayint16 is for signed numbers, and skips leading zeroes).

    Ignore the fact that this 16-bit counter will itself overflow at 65536, i.e., just over 10 minutes.

There are alternative ways of achieving this result. Any IO pins can be used to create pin change interrupts, and also there is an Input Capture unit (see section 15.6 of the datasheet) which is specifically designed for functions like this.

Exercise 2 - sleep

In this exercise we will see how we can reduce the current consumption of the microcontroller utilising the sleep modes.

1. Copy the code from worksheet1 exercise2.c to form worksheet4 exercise2.c, this uses a 1MHz internal clock instead of a crystal.

   Also copy the relevant Makefile.

2. Disconnect the LCD, serial connection, reed switches and associated wiring from the previous exercise. Leave the LED and the crystal connections in place.

3. In order to change the clock source to the internal oscillator, you must issue the `make fuses` command.

   Since the command needs a clock source, you must do this while the crystal and 33pF capacitors are connected. Once the clock has been changed to an internal one, the crystal and 33pF capacitors can be removed.

4. Add a line to exercise2.c as follows to gain access to a number of sleep related functions:

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

Getting the microcontroller to sleep is one half of the problem. Getting it to wake again can be a source of frustration unless care is taken when selecting the sleep mode and the wake-up event. You must carefully consider which events will be used to trigger a wakeup, and make sure that the clocks required to cause those triggers continue to run in the chosen sleep mode.

Refer to section 9 of the datasheet for details.

This exercise will use the Watchdog timer as the wake-up source. It allows wake up from all sleep modes. The watchdog timer's normal purpose is to take control of a device which is not running correctly. In normal use the watchdog receives a 'kick' regularly. Each kick clears a counter which counts up in response to a clock source. If the counter overflows, the watchdog normally issues a reset command to re-initialise the device. An alternative is to issue an interrupt. This exercise will deliberately allow the watchdog timer to overflow and use the resulting interrupt to wake from sleep mode.

To minimise the chances that runaway code will make changes to the watchdog, any change to the watchdog must use a timed sequence. This is described in section 10.8.2

1. Work out a suitable value for the Watchdog Control Register WDTCSR referred to in section 10.9 of the datasheet, to create a watchdog interrupt (not a reset!) at roughly 1 second intervals. For the moment leave WDCE=0 and WDE=0
2. Using the code example WDT_Prescaler_change() right at the end of section 10.8.2 of the datasheet, add a suitable watchdog initialisation function to exercise2.c, plus code to set the master Interrupt Enable bit in SREG.

   Make sure this initialisation code is called before the Master Interrupt Enable bit is set.
3. Add an Interrupt Service Routine ISR(WDT_vect), and set the LED output high when the interrupt occurs.
4. In the main function, use _delay_ms(100) to leave the LED on for 100ms, then use the functions from sleep.h `set_sleep_mode()` and `sleep_mode()` to make the microcontroller go to sleep. Of the sleep modes available, Power Down gives the best power savings.

   You may need to refer to the documentation for <sleep.h> in the avr-libc library avr-libc [http://www.nongnu.org/avr-libc/].
5. Before testing, there is one more factor which makes a huge difference to power consumption, and that is allowing inputs to float, which can increase current consumption significantly. Change the port direction initialisation function to add pull up resistors to all inputs. Make sure the pin for the LED is still assigned as an output.

Apart from a reduction in power consumption, this should behave exactly as per worksheet1 exercise2. Remember that as the clock source has changed, you must issue the `make fuses` command followed by the `make program` command.

Compare the power consumption with sleep mode enabled against that with the sleep_mode() instruction commented out. Adding a multimeter set to a suitable current range in series with the 5V supply, and waiting for the reading to stabilise should be sufficient. You may want to extend the sleep time to 8 seconds in order to leave enough time for the multimeter reading to settle. If the microcontroller is sleeping, the current drawn should be approximately 20uA.

To reduce power consumption still further, it is possible to disable parts of the microcontroller which are not in use.

1. Work out a suitable value for the Power Reduction Register PRR referred to in section 9.11 of the datasheet, to minimise the power drawn as far as possible.
2. Assign the value to the PRR during the initialisation.

   Make sure you do not power down the Watchdog, or the device will never wake from sleep.
3. Try out the revised code, to see if the power reduction is measurable. You might want to extend the watchdog timeout to 8 seconds during testing. This will help you to measure the current consumption.

Important Once you have finished using the multimeter in current mode, move the test leads back to the voltage measurement range. It is very easy to blow the fuse on the meter by putting the leads across a voltage source whilst the leads are plugged in to the current measuring socket.