CG2271 Real Time Operating Systems

Tutorial 2

1. Using Google, Wikipedia or any other source you like, research and explain the following terms:

   a. Address, data and control buses.
   b. I/O bridge.
   c. Address decoding.

   Any sensible answer will do.

2. One disadvantage of memory-mapped I/O is that memory locations that are occupied by I/O cannot be used by the processor to store instructions and data. I.e. parts of the memory range become unavailable.

   An approach that is used in microcontrollers like the Atmel Atmega328p is to extend the memory range so that I/O occupies a separate range of addresses from memory. For example memory might occupy addresses 0-4095, while I/O might occupy addresses 4096-8191. Using what you've understood from Question 1 or otherwise, discuss why this same approach is unsuitable for general processors like the Intel processors found in your desktops and notebooks.

   For cost and power reasons, microcontrollers have very small memory, so that even with a small number of addressing lines, there is still a substantial addressing range left over that can be used for I/O. For example, with just 16 addressing lines (a typical desktop CPU has 32 such lines), there is an addressing range of 64KB. With 32KB of flash and 2 KB of RAM typical of a small microcontroller, there is sufficient addressing range left for over 30,000 devices.

   This is not the case for general purpose CPUs who have to cater for people who want to run large numbers of massive processes requiring a lot of power, and every bit of addressing range is needed just for memory.

3. Given a variable x, write suitable C statements to set bits 3, 5 and 6, and clear bits 0, 1 and 7. If x initially has the value 0xE7, what is the final value in x? Explain all your answers.

We want to set bits 3, 5 and 6, so the mask is 0110 1000 or 0x68. We want to clear bits 0, 1 and 7, so the mask is 0111 1100 or 0x7C. Instructions are:

x |= 0x68; // Sets bits 3, 5 and 6
x &= 0x7C; // Clears bits 0, 1 and 7.

Doing the OR first

7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	1
|	|	|	|	|	|	|	|
0	1	1	0	1	0	0	0
1	1	1	0	1	1	1	1

Doing the AND:

7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1
&	&	&	&	&	&	&	&
0	1	1	1	1	1	0	0
0	1	1	0	1	1	0	0

Final result is 0x6C.

4. Write a bare-metal program (i.e. you are not allowed to use Arduino routines) that flashes an LED connected to digital pin 12 on the Arduino Uno twice a second when a push-button connected to digital pin 11 is not pressed, and four times a second when it is pressed. Your code should not affect the state (0 or 1) or direction (input or output) of any other pin.

   Hint: The AVR library provides a _delay_ms function that delays your program for a specified number of milliseconds. This is not part of the Arduino library and you are allowed to use it.

```
// Arduino pin 11=pin PB3, pin 12 = PB4.

#include <avr/io.h>
#include <util/delay.h>

#define BIT4MASK    0b00010000
#define BIT3MASK    0b00001000

void ledOn()
{
       // Turn on bit 4 of PORTB
       PORTB |= BIT4MASK;
}

void ledOff()
{
       // Turn off bit 4 of PORTB
       PORTB &= ~BIT4MASK;
}
```

```
int main()
{
	// Set PB3 as input (bit 3=0), PB4 as output (bit 4 = 1)
	DDRB |= BIT4MASK;
	DDRB &= ~BIT3MASK;
	unsigned int delay;
	while(1)
	{
		// Note: _delay_ms is called TWICE per cycle, so the delay values
		// below should be HALF of what they’d normally be. I.e. 125 ms
		// and 250 ms for 4 flashes/sec and 2 flashes/sec respectively
		// instead of 250 ms and 500 ms.

		if(PINB & BIT3MASK)
			delay=125;
		else
			delay=250;

		ledOn();
		_delay_ms(delay);
		ledOff();
		_delay_ms(delay);
	}
}
```

5. A common problem with mechanical switches like push-buttons is “switch bounce”, where because of vibrations caused by the switch being pushed and released, the contacts actually open and close many times rather than just once.

    a. Discuss how this would affect the correctness of an embedded device.

       When a user presses a button once, switch bounce may cause the hardware to register multiple button presses, as though the user pressed the button rapidly many times. This can obviously cause incorrect operation. For example if the button was to tell a system how many times to take a sample reading and the user intended to take just one reading, switch bounce can cause the system to errorneously take 5-6 readings instead.

b. Discuss in as much detail as possible how to eliminate this problem through software. You do not need to write any code, perhaps just pseudocode to describe your idea.

The idea of the code is that switch bounce causes RAPID change of switch states from on to off to on etc. By checking that at least a minimum number of milliseconds (30 in this case, can be tuned to balance between performance and eliminating bounce) have passed between changes in switch states, we can avoid the bouncing.

```
// We assume that we are reading pin 11, as per the previous qn.
#define BIT3MASK  0b00001000;

unsigned long curr_time=0, old_time=0;

// ISR for the on-chip timer, configured to trigger every 1 ms
ISR(timer_isr)
{
    // Increment number of milliseconds.
    curr_time++;
}

int main()
{
    int test_button_state, button_state=0;

    // Configure PB3 (pin 11) as input.
    DDRB &= ~BIT3MASK;
    …
    !! Set up the timer to trigger the timer_isr every millisecond
    …

    // Code to read in the button push, with debouncing.

    test_button_state=(PINB & BIT3MASK);
    // Has button state changed?
    if(test_button_state != button_state)
    {
            // Yes, check if at least 30ms has passed since last button state
            // change.

            if(curr_time – old_time > 30)
            {
                    // Yes, record button state and time
                    old_time=curr_time;
                    button_state = test_button_state;
            }
    }


    // The rest of code here
    if(curr_button_state)
            !! Action if  button is pressed.
    else
            !! Action if button is released.
}
```