Digital System Construction

Lab Exercise 6: Microcontrollers

Goals:

The goal of this laboratory exercise is to learn more about microcontrollers, using a development board based on the 32-bit PIC32MX device.

Background:

Microcontrollers are used widely in industry for embedded control and monitoring. Unlike higher-performance microprocessors that are coupled to external memory and peripherals with general-purpose buses, microcontrollers have smaller processors that are interfaced over an internal bus to limited internal memories and built-in peripherals such as ADCs, UARTs, ADCs and general-purpose I/O pins. This makes them a compact, single-chip solution for a wide variety of applications. Computational performance is limited, but this is balanced by low cost and power consumption, which are important considerations especially in portable devices.

Traditionally, microcontrollers were programmed in assembly language to make maximum use of limited memory and CPU power. But modern development tools can produce efficient binary code from compiled languages such as C. Even compact operating systems and BASIC programming environments are available for microcontrollers.

Like many digital systems, the CPU and peripherals are controlled and read using direct read and write operations to control and status registers at fixed addresses. Software libraries are available for most peripheral interfaces, but in this lab you will instead write lower-level code to control peripherals at the register level.

Getting started with MPLAB

MPLAB is a free, full-featured development environment provided by Microchip for programming PIC microcontrollers. It supports a number of compilers including some third party products. For this lab you will work with Microchip's own C32 compiler.

Before beginning, make sure that you also have access to the (large) PIC32MX family reference manual, which contains detailed descriptions of all peripherals and the registers controlling them. You will also need access to the PIC-32MX development board user manual from Olimex. These are available at: http://www.sysf.physto.se/kurser/digsyst/lab6

To get started, Launch MPLAB IDE (type mplab_ide) , and create a new project (e.g. Project1). Choose the following options:

- Use the PIC32MX340F512H device.
- Select the C32 C Compiler and tool suite.

Create a new C source file in your project directory, and then add it to the project you have created. If you choose "C Main File" you will get a template with a main() declaration.

You are now ready to edit your first program.

Your first program

After the title comments, you need to add a pseudo-instruction telling the preprocessor to read a device-specific file before going any further:

```
#include <p32xxxx.h>
```

Based on the device in your project settings, MPLAB will actually include a file called "p32mx360f512h.h" which contains macros for reading and controlling the available registers and their contents.

System configuration

The next step is to set some internal register bits in the PIC that select and configure the system clock. As described in Section 4 of the datasheet, there are a number of different clock sources and options available. For this lab you will use the 8 MHz external crystal oscillator included on the Olimex board, which is connected to the primary oscillator (POSC) input pins of the PIC.

To set these system configuration bits, we include several `#pragma config` lines before the start of your C code.

The description of the system clocks and their settings is in section 4.2. For this lab we would like to drive the 8 MHz external crystal through the POSC pins to produce a 72 MHz system clock. To start, you need to select which oscillator will be used. The OSCCON register (4-1 in the datasheet) includes has three bits (NOSC) for setting this. We are interested in option `011`, which is the primary oscillator with PLL module. In the MPLAB macros, this translates to:

```
#pragma config FNOSC = PRIPLL
```

Next, you need to specify the mode for driving the resonator. Table 4-6 in the datasheet shows the possible operation modes for the POSC. As you can see, the recommended mode for a 3.5-10 MHz resonator is XT, which is a medium-power and medium-frequency mode with medium gain. However, the board manufacturer recommends the higher-gain setting (HS), so you should begin with:

```
#pragma config POSCMOD = HS
```

Going back to the OSCCON register, we now want to synthesize a 72 MHz clock from the 8 MHz input. (9×) To do that, the input to the PLL is first be divided by 2. The PLL itself is set to a multiplication factor of 18, and the final PLL output to the system is divided by 1:

```
#pragma config FPLLMUL = MUL_18, FPLLIDIV = DIV_2, FPLLODIV = DIV_1
```

The internal peripheral bus is also driven by the system clock, but 72 MHz is too high for this bus. Instead we choose a 36 MHz peripheral bus rate by dividing the system frequency by 2: for a 36 MHz bus rate:

```
#pragma config FPBDIV = DIV_2
```

Finally, disable the internal watchdog timer:

```
#pragma config FWDTEN = OFF
```

Note: these settings can be overridden when downloading your design to the microcontroller in MPLAB.

The main() function

Your program must include a main() function, which executes the user-defined part of your code:

```
main()
{
            // your program code goes between the curly brackets.
}
```

Your program can only contain one main() function. In addition, the MPLAB linker will insert a short code segment (`c0`) that performs basic housekeeping before entering the main program. After the main() function has been completed, the processor executes some exit code (`_exit()`) and resets the system. This means that everything starts over again, beginning with the `c0` code.

This is a complete, though not very useful, C program. In the next section you will configure and set some digital I/O pins.

Part 1: Digital I/O pins

The general-purpose digital I/O pins are described and documented in Section 12 of the PIC32MX family data sheet. Some of the pins are multiplexed with several alternate functions. In general, I/O pins can be configured as digital inputs, digital outputs (push-pull or open drain), and for some ports, as analog inputs. Some specific pins are also multiplexed with ports for different peripherals such as UARTs, I2C, SPI, JTAG, etc.

Refer to the datasheet for a detailed list of capabilities for each PORT/pin.

In the diagrams and register tables of Section 12 you will see find two sets of important registers: TRISX, and PORTX, where X is the port (A, B, C…).

The TRISX (for tri-state) registers control the direction of each pin (Input/Output). Each port has a corresponding TRIS register. Setting a bit to 1 configures the corresponding pin as an input. Clearing a bit to 0 configures the pin as an output.

For example, to set all pins 0..7 of PORTE to high you would include the following lines:

```
TRISE = 0x00; // configure the port E pins as outputs
PORTE = 0xff; // set the port E pins to 1
```

To read input values on PORTE, you need to configure the pins as inputs:

```
TRISE = 0xff; // configure the port E pins as inputs
value = PORTE; // read the values of the port E pins
```

If you examine the pin assignments for your device, you will notice that the sixteen pins of PORTB are multiplexed with the analog input pins of port AD1 (see section 22). The analog inputs are enabled by default, so to use PORTB for digital I/O you must first disable the analog input functionality with the AD1 pin configuration register AD1PCFG:

```
AD1PCFG = 0xffff;
```

MPLAB also includes macros to simplify reading and writing to individual bits in each register. To read and write only tto pin 1 of port E you can use commands such as:

```
_RE1 = 1;
```

```
input_bit = _RE1;
```

Assignment: In your main() program write a simple program that reads the value of the button (port D8) and sets the LED accordingly (port F1).

Note: You could make this program in a single statement, and have the `exit()` code restart the program continuously. But this is messy, and a better method is to write an infinite loop:

```
while(1)
    {
      …
    }
```

Now you are ready to compile, link and simulate your design.

Press the Build icon to compile and link your project. The compiler converts your c code and headers and translates them into relocateable object (.o) files. The linker takes the code objects (.o) and libraries (.lib) and assembles them into an executable (.hex) file

Downloading your design to the microcontroller

After you have successfully built the project, connect the USB cable and power adaptor to the PIC MX32 developer board and click on the download icon to program the microcontroller. Test the functionality of your program.

Part 2: Serial communication

Now that you have learned to control a simple peripheral within the MPLAB environment, we turn to a more complex peripheral, one of the built-in UARTs (section 19). There are three important registers for configuring and controlling the UART:

- UXMODE controls the mode register
- UXSTA controls and reads the status register
- UXBRG controls the baud rate of the port.

You will use UART 2. Let's first define the desired settings for each of these three ports.

Begin by studying the contents of the U2MODE register. To enable UART 2, bit 15 (ON) must be set to 1. Bit 11 (RTSMD) needs to be set to 0 if you want to use flow control (with the CTS and RTS pins). Set bit 4 (RXINV) to 0 for the receiver idle state to be '1'. Set bit 3 (BRGH) to 0 to enable standard speed mode (16x baud clock). By setting bits 0-2 to 0 we select 8 bits of data, no parity, and 1 stop bit. In hexadecimal, the contents of U2MODE are then 0x8000. To make the code neater, define the contents of U2MODE with this line before the start of main():

```
#define U_MODE        0x8000     // configure UART2
```

Next, we can turn to the baud rate setting for the UART. The baud rate is defined by dividing the peripheral bus clock (36 MHz) by an appropriate number (see section 19.2). We would like 115200 baud, and we are using a 16x baud clock, so we can derive the U2BRG setting as:

```
#define U_BRG ((36000000/16)/115200 - 1);    // 115200 baud (BREGH=0)
```

Finally, we look at U2STA. Bits 12 and 10 enable the receiver and transmitter, respectively. So to set both, we define the U2STA contents as:

```
#define U_STA         0x1400     // enable transmission and receiving
```

Now you can write the first of three routines for serial communication:

```
// initialize the UART2 serial port
void initU2( void)
{
    U2BRG  = U_BRG;
    U2MODE = U_MODE;
    U2STA  = U_STA;
} // initU2
```

Next, we want to write a routine that writes a character to the UART transmit buffer. To do this, first poll the U2STA register to make sure the buffer is not full (bit 9, UTXBF), and then write the character to the U2TXREG register:

```
// send a character to the UART1 serial port
int putU2( int c)
{
    while ( U2STAbits.UTXBF);   // wait while Tx buffer full
    U2TXREG = c;
    return c;
} // putU2
```

Assignment: write a third routine `char getU2(void)` that polls status register bit 0 (URXDA) until data arrives, then retrieves the byte from register U2RXREG.

In your main program, use your routines to initialize the UART, then receives and transmits bytes. You can make things interesting by, for instance, adding one to the received byte and sending the result back to the host computer. Or have the LED respond to transmitted data.

Download your program to the board. Use a null modem cable and a terminal program such as minicom to test the design.