Experiment 2: ChipKIT$^{\text{TM}}$ Pro and I/O Control

Objectives

The purpose of this experiment is to familiarize the students with the methods of Input/Output (I/O) control of PIC$^{\text{TM}}$32 microcontroller. This includes reading data from input pins and writing to output.

Equipment List

- ChipKIT$^{\text{TM}}$ Pro MX7 processor board with USB cable.
- Microchip MPLAB ® X IDE.
- MPLAB ® XC32++ Compiler.
- MPLAB Harmony Framwwork.

Overview

Microprocessor pins are commonly either configured to be a digital input or digital output, hence are usually referred to as I/O pins. Digital input pins allow microprocessors to receive binary data (one or zero) from their environment. Individual digital output pins can be used to control single function devices such as turning an LED on or off.

The most basic means of communicating to and from a microprocessor is through I/O pins. On the PIC32 microprocessor, I/O pins are grouped in terms of ports, with 16 pins per port, and the collection of bits representing the state of the I/O pins of a designated port itself can be thought of as a word. The pins of each port are identified by a bit position in the form Rp0 to Rp15, where "p" is a letter that represents the particular I/O port on the processor and the numerical value (in the range 0 through 15) is the bit position within the word.

With 16 I/O pins available on a PIC32 port, values can be represented ranging from zero to 65535 (base 10), or, in hexadecimal, zero to 0xFFFF (the leading "0x" indicates a hexadecimal representation of a number where each digit can have one of 16 values [0 through F] while a leading "0b" will be used to indicate a binary representation where each digit can only have one of two values [' 0' or '1']). On the PIC32MX7 microcontroller, **I/O ports are labeled A through G**. Because digital ports are limited to 16 bits, values ranging from zero to 65535 can be directly read from or written to external connections. For example, if RB3 is set high and all other Port B pins are set low, then the processor is outputting a value of eight (0x0008, or in binary, 0b0000 0000 0000 1000).

Input pins can also be used to detect events by sensing switches, button pushes, or some other device that outputs a binary signal, provided that an appropriate voltage level is generated by the sensor device. Multiple event type inputs can be connected to a single 16- bit processor port.

Figure 11 shows a simplified block diagram for a PIC32 I/O pin. The characteristics of each I/O pin are individually controlled by setting bits in four, or in some cases, five registers. The PIC32MX795 processor

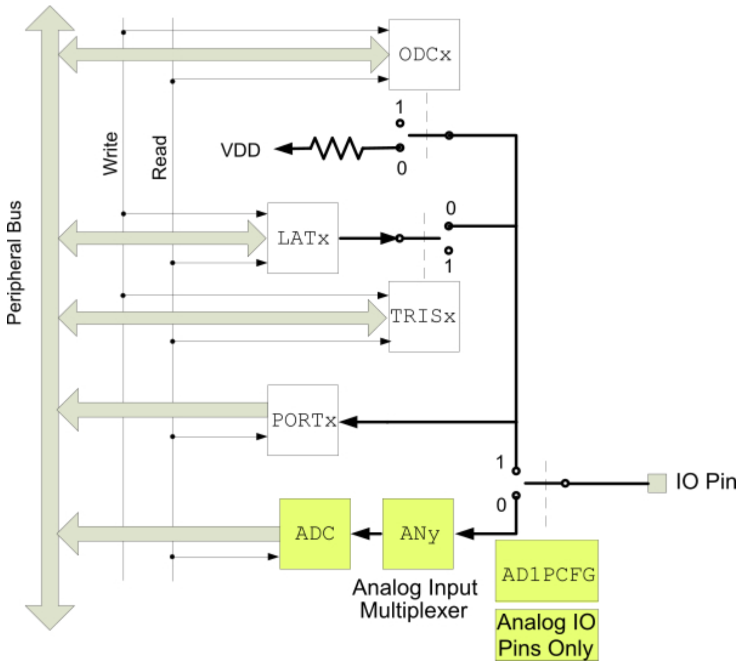

Figure 11: Simplified block diagram for a PIC32 I/O pin.

used on the chipKIT Pro MX7 supports 16 pins that can be used as either an analog input or a digital I/O. By default, when the PIC32 is reset, these pins are set as analog input. The bits in the AD1PCFG register corresponding to the register and a pin number must be set to a '1' if the pin is to be used as a digital input or output.

I/O Pin Registers

TRISi, PORTi, LATi, and ODCi are all 16-bit registers where the designation of i ranges from A through G. The TRISi, LATi, and ODCi registers control the individual I/O pins corresponding to the bit position. The TRIS (tri-state) control determines whether the pin is to be used as an input or an output (I/O pin direction). Setting the bit position in the TRIS register to a '0' (respectively, '1') makes the pin an output (respectively, an input).

The latch (LAT) register is used to control the output level. If the TRIS register is set to configure the I/O pin as an output, a logic '0' written to the LAT register always results in a low voltage level pin. A logic '1' written to the LAT register results in the output pin being either a high voltage level (3.3V) or it results in high impedance, depending on the state of the open drain control (ODC) register. The ODC allows the output to either source up to 25mA at 3.3V or to be in the high impedance state. The default setting (reset condition) for the ODC register is all zeros, which means the outputs can both sink and source current. The state of the pin can still be read by the PORT register.

Built-in LEDs and Buttons

There are 4 LEDs and 3 push buttons on Cerebot MX7cK. LED1 through LED4 are connected to PORTG pins 12 through 15, respectively. BTN1 and BTN2 on the PIC32MX7 are connected to PORTG pins 6 and 7. BTN3is connected to I/O Port A bit 0 and requires disabling the JTAG programmer using the instruction: DDPCONbits.JTAGEN = 0;

Experiment: I/O Control

The only hardware required for this experiment is the ChipKIT^{TM} Pro MX7 processor board. We will be using two of the momentary push buttons as inputs and the four LEDs as outputs.

Part 1: Using I/O Registers

As a first step, you are asked to write a program that turns on all 4 LEDs (LED1-LED4) by using I/O registers only (TRIS, PORT and LAT registers). Don't forget to start by setting the I/O direction of LEDs before outputting data.

All details necessary for writing this code are presented earlier in this manual.

Part 2: Using Harmony Framework: A First Program

Getting familiar to Harmony Projects

As a first step, you are asked to write a simple program to control a LED (LED1 for example) by using a single push button (BTN2 for example). The purpose of this simple task is to get familiar to MPLAB Harmony projects and their file structure. Please follow the following basic steps in writing your code:

- Step 1: Create a Harmony project and configure PIC32MX795F512L, as was shown in Experiment 1.
- Step 2: Add application code to your project. In order to do so, we need to take a deeper look inside the file structure of Harmony projects which was described briefly in Experiment 1.

Source files: main.c

As the name suggests, this file contains the "main" function for an MPLAB Harmony project. It contains calls to the **SYS_Initialize** function, which initializes MPLAB Harmony modules, as well as applications. It also contains the main task execution, which calls tasks for all selected MPLAB Harmony modules, as well as the application task function, **APP_Tasks**.

Here is the basic structure of main.c:

```
int main ( void ){
    /* Initialize all MPLAB Harmony modules, including application(s). */
    SYS_Initialize ( NULL );

    while ( true ){
        /* Maintain state machines of all polled MPLAB Harmony modules. */
        SYS_Tasks ( );
    }
    /* Execution should not come here during normal operation */
    return ( EXIT_FAILURE );
}
```

No modifications needed in your experiment on this function neither the SYS_Initialize function. The configurations and definitions in this function are already done by using the MPLAB Harmony Configurator.

The SYS_Tasks() functions is responsible for the application tasks. It calls the APP_Tasks() function from app.c, which we will modify when we write our own program.

Source files: app.c

This file contains the source code for the MPLAB Harmony application. This file contains the source code for the MPLAB Harmony application. It contains the source code for the MPLAB Harmony application and it implements the logic of the application's state machine. However, it does not call any of the system interfaces (such as the "Initialize" and "Tasks" functions) of any of the modules in the system or make any assumptions about when those functions are called. That is the responsibility of the configuration-specific system files.

In app.c, it contains the **APP_Tasks** function. It defines the logic of the program by the use of a switch statement based on the the current state, as shown in the following code:

```
void APP_Tasks ( void ){
    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:{
            bool appInitialized = true;
            if (appInitialized)
                appData.state = APP_STATE_SERVICE_TASKS;
            break;
        }
        case APP_STATE_SERVICE_TASKS:{
            break;
        }

        /* TODO: implement your application state machine.*/

        /* The default state should never be executed. */
        default:
        {
            /* TODO: Handle error in application's state machine. */
            break;
        }
    }
}
```

As you may notice, the default code performs nothing. It is your task to define the necessary state and state's logic when you write your program. The header file app.h contains the definition of system state enumeration which is defined as follows:

```
    typedef enum{
        /* Application's state machine's initial state. */
        APP_STATE_INIT=0,
        APP_STATE_SERVICE_TASKS

        /* TODO: Define states used by the application state machine. */

} APP_STATES;
```

You can add new states that you find necessary to your code in this enumeration object.

Write Your First Program

1. As a first step, you are asked to write a program that turns on LED1 as long as BTN1 is pressed. When BTN1 is released, LED1 should turn off.

   Note In your code, you could use functions from ports_p32mx795f512l.h. It contains functions to read and write to ports such as:

   ```
   bool PLIB_PORTS_PinGet(PORTS_MODULE_ID index, PORTS_CHANNEL channel,
       PORTS_BIT_POS bitPos);
   ```

BTN2	**BTN1**	**LED4**	**LED3**	**LED2**	**LED1**
OFF	OFF	OFF	OFF	OFF	ON
OFF	ON	OFF	OFF	ON	OFF
ON	OFF	OFF	ON	OFF	OFF
ON	ON	ON	OFF	OFF	OFF

Table 1: Truth table for mapping inputs to outputs

```
void PLIB_PORTS_PinWrite(PORTS_MODULE_ID index, PORTS_CHANNEL channel,
    PORTS_BIT_POS bitPos, bool value);

void PLIB_PORTS_Write(PORTS_MODULE_ID index, PORTS_CHANNEL channel,
int value);
```

where PORTS_MODULE_ID is an enumeration object defined in the same header file. In you code, you can use PORTS_ID_0 as its value. The PORTS_CHANNEL and PORTS_BIT_POS are used to specify which pin you are about to read or write to. Available channel on PIC32MX795L512F are from A to G and available bit positions are from 0 to 15. Back to the documentation of this PIC, BTN1 is at RG06, which means PORT_CHANNEL_G and PORTS_BIT_POS_6. Refer to ports_p32mx795f512l.h for more information about other available definitions.

2. Step two: Modify your code so as to use BTN1 and BTN2 to control the four LEDs (LED1 - LED4). based on the action described in Table 1.

Note: to run the program on the Chipkit MX7ck, you have to click on the **Make and Program Device Main project** button. This action opens a **Hardware Tools** window which allows you to choose a suitable debugger for your hardware. The debugging tool provided by Digilent for the ChipKIT^{TM} Pro MX7 processor board appears under the **Other Tools** folder. If a ChipKIT^{TM} Pro MX7 processor is connected to the PC via the USB cable and the power to the board is turned on, the board's serial number will show on the list. Note: After the project has been built, if the project launched when the PC is connected to a different ChipKIT^{TM} Pro MX7 processor board, MPLAB ® X will prompt you to approve the connection to the new board.

Part 3: A simple counter

In this part, you are asked to write a program to develop a 4-bit counter by using the 4 LEDs of the board. The displayed values are from 0 to 15. Choose a counting speed that you find suitable.