CpE5160 Experiment #1
General Purpose I/O

Introduction

The purpose experiment is to create some functions for controlling output pins and reading input pins. These simple I/O functions will create general purpose functions that can be used to create chip selects and other controls for accessing the SD card and MP3 decoder in later experiments. The experiment will also introduce methods of organizing source code and using pointers to access peripherals and variables.

Creating I/O Functions

There are four basic I/O functions needed, initializing an input pin, reading an input pin, initializing an output pin, and writing a value to an output pin. I recommend placing the GPIO output functions and GPIO input functions in two separate files, but this is not required. The programmer may choose to use separate set output and clear output functions. It is recommended to use a pointer to reference each port and a struct to overlay the GPIO special function registers. The pointer must be declared as “volatile” to indicate that the port registers can change outside of the normal program flow. Each port has three registers. The PINx register at offset 0 from the base address and is used to read the value on the pins. The DDRx register at offset 1 from the base address is the Data Direction Register which is used to set pins as inputs (‘0’) or outputs (‘1’). The PORTx at offset 2 from the base address is used to set (‘1’) or clear (‘0’) output values. It can also be used to enable (‘1’) or disable ‘(0’) the internal pull-up when a pin is defined as an input.

```
typedef struct port_regs
{
   volatile uint8_t PIN_REG;
   volatile uint8_t DDR_REG;
   volatile uint8_t PORT_REG;
} GPIO_port_t;
```

It is recommended that the value being written to the GPIO special function registers be a bit mask byte, such as #define LED0_pin (1<<7), be used instead of a bit number, such as #define LED0_bit (7), to specify which bits are referenced. This helps in two ways. First, it allows the programmer to control multiple pins at once.

```
#define LED0_pin (1<<7)   // 0b10000000
#define LED1_pin (1<<5)   // 0b00100000

#define LEDS (LED0_pin|LED1_pin)  // 0b10100000
```

Second, a bit number indicates that a bit value must be shifted instead of a predefined number and will require extra clock cycles to execute.

```
#define LED0_pin (7)
…
DDRC|=(1<<LED0_pin);  requires a left shift and then OR
```

The macros for setting a bit or clearing a bit are generally considered the most efficient method for controlling general purpose I/O, but they are not ANSI C standard instructions and are therefore not very portable to other microcontrollers and compilers. The macros or writing directly to a GPIO register are the quickest methods for accessing an I/O pin. So, if speed is most important in an application (every clock cycle counts type of application), then these methods may need to be used.

```
#define set_bits_macro(port,mask) ((port) |= (mask))
#define clear_bits_macro(port,mask) ((port)&=(~(mask)))
```

Using a pointer to a struct that is an overlay of special function registers is commonly used among microcontroller manufacturers and compliers for defining access to these registers. This makes source code written in this manner portable and reusable.

Procedure

The following equipment will be required for this part of the experiment:

- ATMEGA324PB Xplained development board
- Micro-B USB Cable
- Files from Canvas:
    - Example_Project
- PC with
    - Atmel Studio 7
    - Terminal Program (For Example: Putty)

1) (2pts) Create a new executable C project for this experiment or modify the example project. This will automatically create a main.c source code file. Any project should have a board.h file to describe your hardware. Make sure the defined values F_CPU and OSC_DIV defined in board.h match with your circuit which has an oscillator frequency of 16MHz.

2) (8pts) Create a new header file and C source code file to contain the device driver functions for the general-purpose outputs (GPIO_Output_Init(), GPIO_Output_Set() and GPIO_Output_Clear()). The functions should use a pointer to specify which port and a bit-mask to specify the bits. As an alternative to the GPIO_Output_Set() and GPIO_Output_Clear() functions, the programmer may choose to have an GPIO_Output_Control() function with a parameter to specify the bit value.

3) (6pts) Create a hardware abstraction layer header file and C source code file to contain functions for controlling the LEDs on the ATMEGA324PB development

board. The header file should define public constants for the LED port addresses and the pin numbers for LED0 on PC7, LED1 on PB3, LED2 on PE4 and LED3 on PA7. There should be a public function for initializing an LED pin as an output and set as ‘1’ so it is off (all the LEDs are active low). There should also be functions to switch an LED on and switch the LED off. All these functions should call the device drivers from the previous step to control the LEDs.

4) (4pts) Use the hardware abstraction layer functions created in the previous step to initialize all the LEDs. The initialization area is the code at the beginning of the main function before the endless “super-loop.” Use the LED on and off functions along with a delay function in the while(1) loop to create some blinking LED pattern to verify that the code is working. I recommend including the library <util/delay.h> and calling the function _delay_ms(number of milliseconds). This library requires the constant F_CPU to be set correctly or the delay will not be accurate (i.e.; the F_CPU constant must be declared in “board.h” before <util/delay.h>, so place #include “board.h” first in the source code).

5) (8pts) Create a new header file and C source code file to contain the device drivers for the GPIO Input functions or add them to your existing GPIO source code. There should be a GPIO_Input_Init() function. There should be a parameter for the port, a pin mask, and then one to enable or disable the internal pull-up. A second function, GPIO_Input_Read_Pin(), should read the value of the pin and return a ‘1’ or ‘0.’ Note that a pin mask type value should not be returned. If the programmer wants to have a function that returns a pin mask value, I suggest a GPIO_Input_Read_Port() function that can read and entire port and return multiple pin masked values.

6) (6pts) Add to the hardware abstraction layer header file and C source code file to include functions for reading the switches on the ATMEGA324PB development board. The header file should define public constants for the switch port addresses and the pin numbers for SW0 on PC6, SW1 on PB2, SW2 on PA4 and SW3 on PA5. There should be a public function for initializing a switch pin as an input with the pull-up resistor enabled. The switches are active low, meaning that they will read as ‘0’ when the switch is pressed. The programmer may wish to have a Read_Switch() function that includes some simple debouncing such as read the GPIO input pin, if it is ‘0,’ then wait for 75ms, and then read the GPIO input pin again. If it is still ‘0,’ then the switch is pressed. If you do not include debouncing, then the action that the switch takes will need to include some delay so that actions are not repeated while the switch is pressed.

7) (6pts) Create an application that initializes the switches and LEDs with the LEDs off as the initial state. The application should read each of the switches. If SW0 is pressed once, then LED0 should switch on. Pressing SW0 again should cause LED0 to flash 0.1s on and 0.4s off and keep repeating this pattern. If SW0 is pressed again, then LED0 should switch off where it can be switched back on with a fourth switch press and so on. Note: If you only read the switch once

during the blinking pattern, you may need to hold the switch until the LED switches off. This is acceptable for this first experiment.

Repeat the same switch press sequence for the other switches and LEDs. You can make the other LEDs blink at different rates if you want. It acceptable that if one switch is blinking that no other LEDs can blink, and no other switches can be read. In other words, to make the program simple you can be in a while loop reading on switch and controlling one LED when it is blinking. The simple embedded operating system and other real-time operating system techniques will be discussed later to allow more complex control of reading switches and blinking LEDs (among other actions).

8) A demonstration of this code is not needed. Place all source code and header files into a compressed folder (zip file) and post it to canvas or e-mail it to me at youngerr@mst.edu. Make sure that if you are working with a group, place your names in the comment headers. Only one person in the group needs to post the source code files.

Grading:

Functionality:	Code works as described in the assignment: Registers set as needed and actions occur as specified; Questions answered and calculations shown.	Points awarded as specified in the steps marked graded. (40 points)
Compile Errors:	If submitted code does not compile, then a minimum of 25% of the functionality points will be deducted. More may be deducted for multiple errors.	Minimum of 25% of functionality point deducted.
Organization:	Separating device drivers (general-purpose I/O) and application code.	10 points
Readability:	Source code is well commented. Descriptive names are used for functions, constants and variables.	10 points
Correctness	No patches or work-arounds are used in the source code to get the correct functionality.	10 points