Chapter 1

Digital Labs

1.1 Lab 1 - The CPU and General Purpose I/O

Cypress' PSoC (Programmable System-on-Chip) is a family of mixed-signal arrays based on an integral microcontroller and configurable analog/digital peripherals. It is a software configurable, mixed-signal array with a built-in 8-bit Harvard architecture core called the M8C. PSoC has three separate memory spaces: paged SRAM for data, Flash memory for instructions/fixed data, and I/O Registers for controlling and accessing the configurable logic blocks and functions. At power up it is configured configuration by loading instructions from the built-in Flash. One of the most important features of PSoC is the variety of ways its I/O pins can be used and configured, i.e., the manner in which the CPU interacts with the configurable user modules and the I/O pins.

Objective: This Lab will serve as an introduction to the PSoC microcontroller (CPU), designated the M8C, and the interface to its I/O ports. Note that this Lab is divided into three parts, Lab 1A, Lab 1B, and Lab 1C respectively. It is suggested that you complete complete each Lab before proceeding to the next.

Upon successful completion of this Lab you will:

- have used PSoC Designer to create a new project,
- understand the basic operation and functionality of the CPU,
- understand what system parameters affect CPU performance,
- have declared global variables in assembly language,
- have calculated cycle time for a CPU control loop,
- understand how to read and write an I/O port,
- have configured I/O port pins for different drive modes,
- have used shadow registers to isolate input/output interaction,
- have configured an I/O port pin to read a momentary switch,

and,

- have written a debounce routine to correctly read this switch.

Assumptions:

- PSoC Designer and PSoC Programmer are installed on your computer.

Required Materials:

- CY3210 PsoCEval1 Board[1] and power supply.
- Breadboard wire.

Required Equipment:

- Oscilloscope.

Related Reference Material:[2]

- Cypress Technical Reference Manual.
- Cypress Assembly Language User Guide.
- Cypress Tele-Training Video Module 1: Introductory Module.
- Cypress Tele-Training Video Module 2: Getting Started Designing.

1.1.1 Lab 1A - CPU

Step 1. Start a New Project.

- Load PSoC Designer and click on "Start a new project".
- Select the "base part" to be a **CY8C27443-24PXI**. Name the Project "**Lab1A**". (*Unless explicitly expressed, all Labs use CY8C27443-24PXI as the base part).*
- Select the **Assembler** Option.
- Navigate to the **Device Editor** and then switch to the **Interconnect View**, as shown in Figure 1.1.

Step 2. Set Necessary Global Parameters.

The Global Resources window, located in the upper left of the PSoC Designer screen contains 20 parameters as shown in Figure 1.2, only two of which, viz., **CPU_Clock**, and **Supply Voltage**, affect CPU operation. Eight of these 20 parameters, are CPU clock selections. SysClk/1 (24 MHz) should only be used when the supply voltage is guaranteed to be greater than 4.75 volts.

There are two possible voltage settings available. For the 3.3 V setting the fastest allowable CPU clock is $SysClk/2(12MHz)$ as shown in Figure 1.3.

Unless stated otherwise, all Lab projects will have these parameters set to 5.0 V and 12 MHz, as shown in Figures 1.4 and 1.3, respectively.

Step 3. Set the Drive Modes for Port 1 Pins.

- The Port Selection window is located in the lower left of the PSoC Designer screen. Set the port 1 pins to have a strong drive.

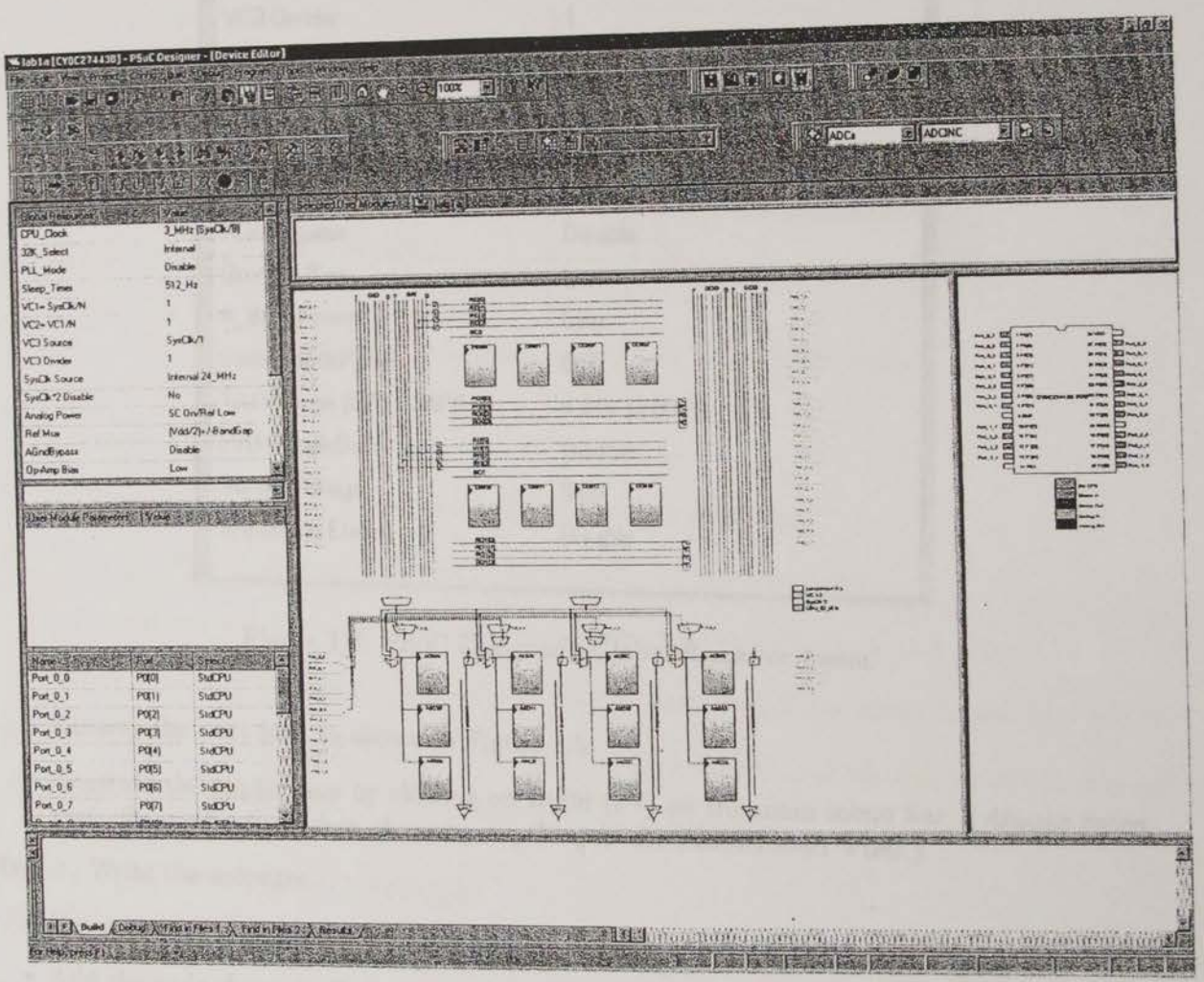

Figure 1.1: PSoC Designer's Interconnect View.

Global Resources	Value
CPU_Clock	3_MHz (SysClk/8)
32K_Select	Internal
PLL_Mode	Disable
Sleep_Timer	512_Hz
VC1= SysClk/N	1
VC2= VC1/N	1
VC3 Source	SysClk/1
VC3 Divider	1
SysClk Source	Internal 24_MHz
SysClk*2 Disable	No
Analog Power	SC On/Ref Low
Ref Mux	(Vdd/2)+/-BandGap
AGndBypass	Disable
Op-Amp Bias	Low
A_Buff_Power	Low
SwitchModePump	OFF
Trip Voltage [LVD (SMP)]	4.81V (5.00V)
LVDThrottleBack	Disable
Supply Voltage	5.0V
Watchdog Enable	Disable

Figure 1.2: PSoC Designer's Global Resource menu.

- Rename the port pins as shown in Figure 1.5.
- Generate the application by clicking on Build (F7) on the main menu bar. (*Always regenerate the application when changing anything in the Interconnect View.*)

Step 4. Write the software.

- Navigate to the Application Editor and open **main.asm**. It is shown in Figure 1.6.
- Add the code shown in Figure 1.7.

Exercise 1A-1: What does this code do?

Exercise 1A-2: What is the loop cycle time (in CPU cycles?)

Exercise 1A-3: For a 12 MHz CPU clock, what is each pin's output frequency?

Step 5. Download code and Run.

- Build this application and verify that there are no errors.

[1] Note that throughout this manual, references to "Eval1" are references to Cypress' CY3210 PSoCEval1 evaluation board.

[2] Available at www.cypress.com

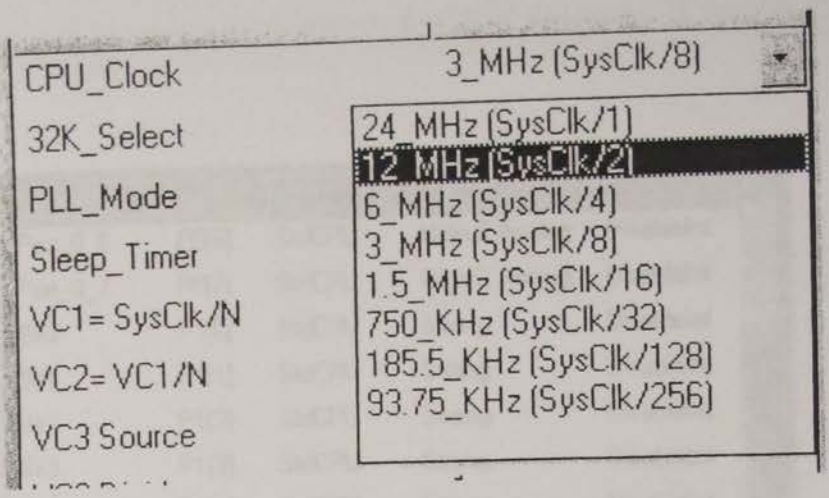

Figure 1.3: PSoC Designer's Global Resource Clock sub menu.

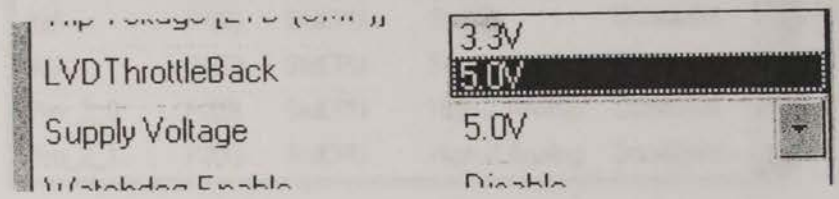

Figure 1.4: PSoC Designer's Supply voltage sub menu.

- Download the program to the Eval1 board and run it.
- Use an oscilloscope to view the waveform found on each pin.

Exercise 1A-4: Do these measurements agree with your calculations?

- Navigate back to the Interconnect View and change the CPU clock to 3 MHz.
- Regenerate the application, rebuild the project, download to the Eval1 board, and run it.

Exercise 1A-5: With the CPU clock now set to 3 MHz, how should the pin signals to change?

Exercise 1A-6: Do your oscilloscope measurements verify this?

1.1.2 Lab 1B - GPIO Output

Step 1. Make a copy of Lab1A.

- Open project **Lab1A**.
- Navigate to the File menu and select **Save Project As**.
- Set the **New Project Name** value to **Lab1B**.

Step 2. Set the Drive Modes for Port 1 Pins.

- The Port Selection window is located in the lower left of the PSoC Designer screen as shown in Figure 1.1. Change the port 1 pin drive selections from **Strong** to **PullUp** as shown in Figure 1.8.

Name	Port	Select	Drive	Interrupt
Port_0_6	P0[6]	StdCPU	High Z Analog	DisableInt
Port_0_7	P0[7]	StdCPU	High Z Analog	DisableInt
Bit0	P1[0]	StdCPU	Strong	DisableInt
Bit1	P1[1]	StdCPU	Strong	DisableInt
Bit2	P1[2]	StdCPU	Strong	DisableInt
Bit3	P1[3]	StdCPU	Strong	DisableInt
Bit4	P1[4]	StdCPU	Strong	DisableInt
Bit5	P1[5]	StdCPU	Strong	DisableInt
Bit6	P1[6]	StdCPU	Strong	DisableInt
Bit7	P1[7]	StdCPU	Strong	DisableInt
Port_2_0	P2[0]	StdCPU	High Z Analog	DisableInt
Port_2_1	P2[1]	StdCPU	High Z Analog	DisableInt

Figure 1.5: Port Drives

main.asm

```
;-----------------------------------------------------------------------------
; Assembly main line
;-----------------------------------------------------------------------------

include "m8c.inc"        ; part specific constants and macros
include "memory.inc"     ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"    ; PSoC API definitions for all User Modules

export _main

_main:

    ; Insert your main assembly code here.

.terminate:
    jmp .terminate
```

Figure 1.6: Default source listing for main.asm.

main.asm

```
;-----------------------------
; Lab1A Control Software
;-----------------------------

include "m8c.inc"        ; part specific constants and macros
include "memory.inc"     ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"    ; PSoC API definitions for all User Modules

export _main

_main:
loop:
   mov   A,reg[PRT1DR]
   inc   A
   mov   reg[PRT1DR],A
   jmp loop
```

Figure 1.7: Modified source listing for main.asm.

Bit0 = _____ Bit1 = _____
Bit2 = _____ Bit3 = _____
Bit4 = _____ Bit5 = _____
Bit6 = _____ Bit4 = _____

Writing to a port causes the port register to be set. The logic level at the pin is dependent on the drive register value, the drive mode, and external circuitry connected to the pin. Reading from a port causes the data to be taken directly from the pins. When configured as pull-ups, the outputs have a strong impedance drive to logic low and a resistive drive to logic high. When driven high these pins can be pulled low externally.

Step 3. Add the necessary jumpers to the Eval1 board.

- Use jumper wire to short Bit0 to Bit6 on the Eval1 board. These two pins are "wire -ANDed". Driving either pin low causes both pins to be pulled low.

Exercise 1B-1: Given this information, what should the waveform on each pin be?

Step 4. Download the code and run it.

- Set the **CPU_Clock** global parameter to **12 MHz**.
- Regenerate the application, rebuild the project, download it to the board, and run it.
- Use an oscilloscope to view the waveform on each pin.

Exercise 1B-2: Do your observations agree with your predictions?

Step 5. Use a shadow register.

- Navigate to the Application Editor and open **main.asm** (shown in Figure 1.9).

Name	Port	Select	Drive	Interrupt
Port_0_6	P0[6]	StdCPU	High Z Analog	DisableInt
Port_0_7	P0[7]	StdCPU	High Z Analog	DisableInt
Bit0	P1[0]	StdCPU	Pull Up	DisableInt
Bit1	P1[1]	StdCPU	Pull Up	DisableInt
Bit2	P1[2]	StdCPU	Pull Up	DisableInt
Bit3	P1[3]	StdCPU	Pull Up	DisableInt
Bit4	P1[4]	StdCPU	Pull Up	DisableInt
Bit5	P1[5]	StdCPU	Pull Up	DisableInt
Bit6	P1[6]	StdCPU	Pull Up	DisableInt
Bit7	P1[7]	StdCPU	Pull Up	DisableInt
Port_2_0	P2[0]	StdCPU	High Z Analog	DisableInt
Port_2_1	P2[1]	StdCPU	High Z Analog	DisableInt

Figure 1.8: PSoC Designer's port selection window.

- Add the code shown in Figure 1.10.

Note:

1. There is code added to allocate a byte of RAM for a variable. This particular variable is named **bShadow**.

2. The **export** statement makes **bShadow** a global variable. It is available to all other code files in this project. The format for the contents of bShadow is [**bShadow**].

The state of the output register is now stored in a variable. Doing this breaks the I/O "input/output" interaction.

Exercise 1B-3: Now that the input output interaction has been broken, what should the waveform of each output look like?

Exercise 1B-4: What is the loop cycle time (in CPU cycles?)

Step 6. Download and Run.

- Rebuild the project, download it to the board, and run it.
- Use an oscilloscope to view Bit6 while triggering on Bit7.

Exercise 1B-5: Do your observations agree with your predictions?

1.1.3 Lab 1C - GPIO Input

Step 1. Create a New Project.

- Name it **Lab1C**.
- Use the standard Lab settings for part type and settings.

main.asm

```
;-----------------------------
; Lab1A Control Software
;-----------------------------

include "m8c.inc"        ; part specific constants and macros
include "memory.inc"     ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"    ; PSoC API definitions for all User Modules

export _main

_main:
loop:
   mov   A,reg[PRT1DR]
   inc   A
   mov   reg[PRT1DR],A
   jmp loop
```

Figure 1.9: Source listing for main.asm

main.asm

```
;-----------------------------
; Lab1B Control Software
;-----------------------------

include "m8c.inc"        ; part specific constants and macros
include "memory.inc"     ; Constants & macros for SMM/LMM and Compiler
include "PSoCAPI.inc"    ; PSoC API definitions for all User Modules

export bShadow
area bss(RAM)
    bShadow:  BLK  1     ; Reserve a RAM location named bShadow
area text(ROM,REL)

export _main
_main:
   mov   [bShadow],0
loop:
   inc   [bShadow]
   mov   A,[bShadow]
   mov   reg[PRT1DR],A
   jmp   loop
```

Figure 1.10: Source listing modified to support shadow register use.

Step 2. Set the Drive Modes for Pins and Make External Connections.

- Set the five port pins as shown in Table 1.1.

Table 1.1: Drive Modes

Name	Port	Select	Drive
SwitchInput	P1[0]	StdCPU	PullDown
LedOut1	P1[1]	StdCPU	Strong
LedOut2	P1[2]	StdCPU	Strong
LedOut3	P1[3]	StdCPU	Strong
LedOut4	P1[4]	StdCPU	Strong

Step 3. Add jumpers to the Eval1 board.

- Use jumper wire to make the following connections.

1. P10 to SW
2. P11 to LED1
3. P12 to LED2
4. P13 to LED3
5. P14 to LED4

This configuration is shown in Figure 1.11.

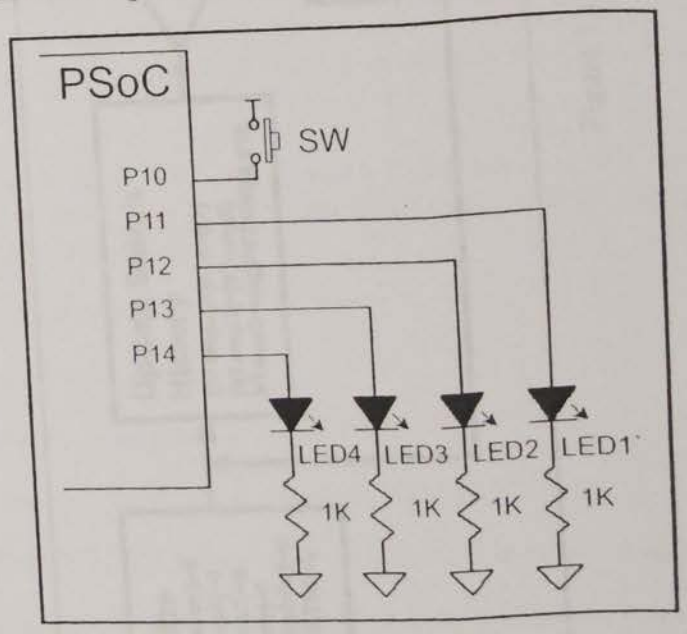

Figure 1.11: EVAL1 Board LED connections.

P10 (SwitchInput) is configured with a "pull down" drive mode. When this port pin's data register is set to zero the output appears as a resistance to ground. When the external switch is pressed the pin is pulled high. Although configured as an output, this pin is used as an input, as long as the data register bit remains zero.

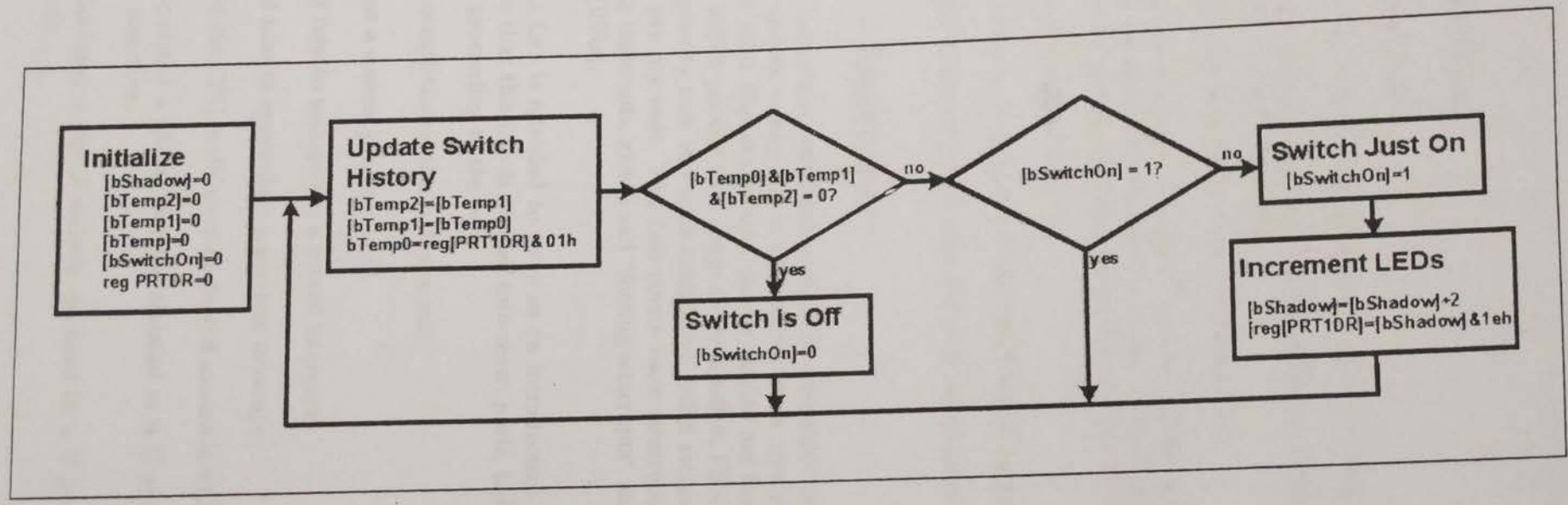

Figure 1.12: Switch debounce algorithm.

- Generate the Application.

Step 4. Write Control Code.

- Write an assembly control program that does the following.

  1. Declares five global single byte variables named, [**bShadow**], [**bTemp2**], [**bTemp1**], [**bTemp0**], and [**bSwitchOn**].
  2. Implements the flow chart shown in Figure 1.12.

This algorithm contains a switch "debounce" routine. Three consecutive readings of the switch must be high before the switch is considered to be on. The LEDs are incremented, via the shadow register, only when the switch state goes from off to on (low to high).

Step 5. Download the code and run it.

- Rebuild the project, download it to the Eval1 board, and run it.
- Verify that the LEDs increment once for each switch press.

1.2 Lab 2 - Interrupts

Interrupts are an important consideration in most embedded system applications and interrupt support allows a system to respond to asynchronous, or unscheduled, events based on relative priorities, i.e., the most important tasks can be carried out first. Thus if a lower priority task is running and a higher priority task needs to be handled, PSoC can stop the running task, address the higher priority task and upon completion either return to the original task, or address the next highest priority task. This Lab covers basic interrupt handling concepts, e.g., posted interrupts, ending interrupts, global and "forcing interrupts" and the creation of small interrupt service routines (ISRs).

Objective: This Lab is intended to serve as an introduction to the interrupt handler for the PSoC CPU. Note that this Lab is divided into seven parts, Lab 2A through Lab 2G. Complete each part before proceeding to the next.

Upon successful completion of this Lab you will:

- have written a control program in C,
- understand how to manipulate a posted interrupt,
- understand how to manipulate a pending interrupt,
- have placed the CPU in the sleep mode and awaken it with a sleep timer pending interrupt,
- have manipulated a global variable declared in a C program with an interrupt handler written in assembly,
- have manipulated a global variable declared in a C program with an interrupt handler written in C,

and,

- have placed a small ISR in the interrupt vector space.

- Generate the Application.

Step 4. Write Control Code.

- Write an assembly control program that does the following.
  1. Declares five global single byte variables named, [**bShadow**], [**bTemp2**], [**bTemp1**], [**bTemp0**], and [**bSwitchOn**].
  2. Implements the flow chart shown in Figure 1.12.

This algorithm contains a switch "debounce" routine. Three consecutive readings of the switch must be high before the switch is considered to be on. The LEDs are incremented, via the shadow register, only when the switch state goes from off to on (low to high).

Step 5. Download the code and run it.

- Rebuild the project, download it to the Eval1 board, and run it.
- Verify that the LEDs increment once for each switch press.

1.2 Lab 2 - Interrupts

Interrupts are an important consideration in most embedded system applications and interrupt support allows a system to respond to asynchronous, or unscheduled, events based on relative priorities, i.e., the most important tasks can be carried out first. Thus if a lower priority task is running and a higher priority task needs to be handled, PSoC can stop the running task, address the higher priority task and upon completion either return to the original task, or address the next highest priority task. This Lab covers basic interrupt handling concepts, e.g., posted interrupts, ending interrupts, global and "forcing interrupts" and the creation of small interrupt service routines (ISRs).

Objective: This Lab is intended to serve as an introduction to the interrupt handler for the PSoC CPU. Note that this Lab is divided into seven parts, Lab 2A through Lab 2G. Complete each part before proceeding to the next.

Upon successful completion of this Lab you will:

- have written a control program in C,
- understand how to manipulate a posted interrupt,
- understand how to manipulate a pending interrupt,
- have placed the CPU in the sleep mode and awaken it with a sleep timer pending interrupt,
- have manipulated a global variable declared in a C program with an interrupt handler written in assembly,
- have manipulated a global variable declared in a C program with an interrupt handler written in C,

and,

- have placed a small ISR in the interrupt vector space.

- Generate the Application.

Step 4. Write Control Code.

- Write an assembly control program that does the following.
  1. Declares five global single byte variables named, [**bShadow**], [**bTemp2**], [**bTemp1**], [**bTemp0**], and [**bSwitchOn**].
  2. Implements the flow chart shown in Figure 1.12.

This algorithm contains a switch "debounce" routine. Three consecutive readings of the switch must be high before the switch is considered to be on. The LEDs are incremented, via the shadow register, only when the switch state goes from off to on (low to high).

Step 5. Download the code and run it.

- Rebuild the project, download it to the Eval1 board, and run it.
- Verify that the LEDs increment once for each switch press.

1.2 Lab 2 - Interrupts

Interrupts are an important consideration in most embedded system applications and interrupt support allows a system to respond to asynchronous, or unscheduled, events based on relative priorities, i.e., the most important tasks can be carried out first. Thus if a lower priority task is running and a higher priority task needs to be handled, PSoC can stop the running task, address the higher priority task and upon completion either return to the original task, or address the next highest priority task. This Lab covers basic interrupt handling concepts, e.g., posted interrupts, ending interrupts, global and "forcing interrupts" and the creation of small interrupt service routines (ISRs).

Objective: This Lab is intended to serve as an introduction to the interrupt handler for the PSoC CPU. Note that this Lab is divided into seven parts, Lab 2A through Lab 2G. Complete each part before proceeding to the next.

Upon successful completion of this Lab you will:

- have written a control program in C,
- understand how to manipulate a posted interrupt,
- understand how to manipulate a pending interrupt,
- have placed the CPU in the sleep mode and awaken it with a sleep timer pending interrupt,
- have manipulated a global variable declared in a C program with an interrupt handler written in assembly,
- have manipulated a global variable declared in a C program with an interrupt handler written in C,

and,

- have placed a small ISR in the interrupt vector space.

Assumptions:

- All previous Lab assumptions.
- Compiler enabled for your development system.

Required Materials:

- CY3210 PSoCEval1 Board.
- Breadboard wire.

Required Equipment:

- None.

Related Reference Material:

- All previous Lab reference materials.
- Cypress C Language Compiler Users Guide.

The interrupt controller provides a mechanism for several different hardware resources in the PSoC Mixed Signal Array to change program execution to a new address without regard to the current task being performed. A block diagram is shown in Figure 1.13.

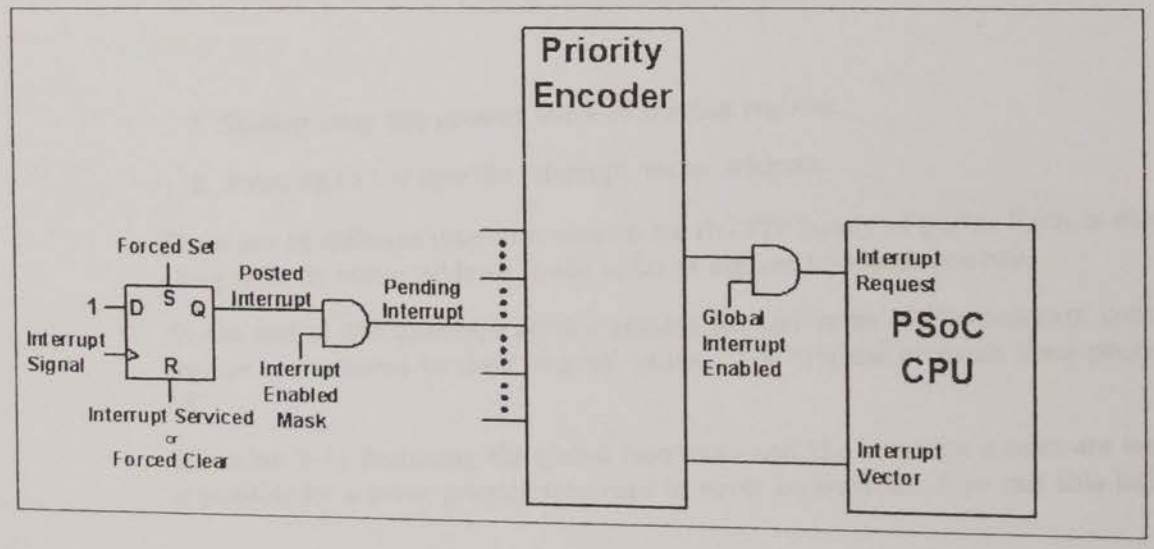

Figure 1.13: Interrupt handling.

For the CPU to recognize a particular resource's interrupt the following must be true:

1. Its Interrupt Enabled Mask must be set.
2. It must be the highest priority of all the pending interrupts.
3. The Global Interrupt Enable must be set.

The CPU responds to this request by:

1. Storing away the present value of the program counter.

Table 1.2: Interrupt Vector Table

Interrupt Source	Vector Location
Hardware Reset	0x0000
Supply Monitor	0x0004
Analog Column 0	0x0008
Analog Column 1	0x000c
Analog Column 2	0x0010
Analog Column 3	0x0014
VC3 Clock	0x0018
GPIO	0x001c
Digital Block 00	0x0020
Digital Block 01	0x0024
Digital Block 02	0x0028
Digital Block 03	0x002c
Digital Block 10	0x0030
Digital Block 11	0x0034
Digital Block 12	0x0038
Digital Block 13	0x003c
I2C	0x0060
Sleep Timer	0x0064

2. Storing away the present value of the flag register.

3. Jumping to the specific interrupt vector address.

There are 18 different interrupt sources for the 27x family of parts. Each is shown in Table 1.2, along with its vector address in the order of highest to lowest priority.

At the end of the interrupt service routine the old value of the program counter and the flag register are restored to their original values. The original program then proceeds where it left off.

Exercise 2-1: Assuming the global interrupts and the interrupt masks are enabled properly, it is possible for a lower priority interrupt to never be serviced. How can this be?

1.2.1 Lab 2A - Posted Interrupts

Access to the pending interrupts is determined by the **INT_CLRx** registers. Control of the posted interrupt for the sleep timer is determined by **Bit6** of the **INT_CLR0** register. When this bit has a value of one, there is a posted sleep timer interrupt. Setting this bit to zero clears its posted interrupt.

Step1. Create New Project.

- Name it **Lab2A**.
- Select the C option.
- Use the standard Lab settings for part type and settings.

- Navigate to the Interconnect View and set the **Sleep_Timer** global parameter to **1_Hz.** *(This will cause a sleep timer pending interrupt to be generated once a second.)*

- Set the four port pins as shown in Table 1.3.

Table 1.3: Port Drive Settings

Name	Port	Select	Drive
LedOut1	P1[0]	StdCPU	Strong
LedOut2	P1[1]	StdCPU	Strong
LedOut3	P1[2]	StdCPU	Strong
LedOut4	P1[3]	StdCPU	Strong

Step 2. Add jumpers to the Eval1 board.

- Use jumper wires to make the following connections:

  1. P10 to LED1
  2. P11 to LED2
  3. P12 to LED3
  4. P13 to LED4

- Generate the Application

Step 3. Write the software.

- Navigate to the Application Editor and open **main.c** as shown in Figure 1.14.

main.c

```
//-------------------------------------------------------------------------
// C main line
//-------------------------------------------------------------------------

#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules


void main()
{
    // Insert your main routine code here.
}
```

Figure 1.14: main.c prototype.

- Add the code shown in Figure 1.15.

main.c

```
//-----------------------------
// Lab2A Control Software
//-----------------------------
#include <m8c.h>          // part specific constants and macros
#include "PSoCAPI.h"      // PSoC API definitions for all User Modules

extern unsigned char bShadow = 0;
void main(){
   PRT1DR = 0;
   while(1){
      while((INT_CLR0 & 0x40) == 0); //wait till set
      INT_CLR0 = INT_CLR0 & ~0x40;   //Clear it
      bShadow++;
      PRT1DR = bShadow;
   }
}
```

Figure 1.15: Lab 2A control software.

Note that:

1. **bShadow** is defined to be a global single byte variable.
2. **INT_CLR0** is the register used to monitor and clear the sleep timer posted interrupt.

Exercise 2A-1: What does this code do?

Step 4. Download the code and run it.

- Build this project and verify that there are no errors.
- Download the program to the Eval1 board and run.
- Verify correct operation by observing the LEDs.

Exercise 2A-2: Do the LEDs operate as you predicted in Exercise 2?

- Navigate back to the Interconnect View and change the sleep timer parameter to **8_Hz**.

Exercise 2A-3: With the sleep timer parameter now at 8_Hz, what effect should this have on the LEDs' outputs?

- Regenerate the application, rebuild the project, download it to the Eval1 board, and run it.

Exercise 2A-4: Does your observation confirm your Exercise 2A-4 prediction?

1.2.2 Lab 2B - Pending Interrupts

Step 1. Make a copy of Lab2A.

- Open **Lab2A**.
- Save Project As **Lab2B.**
- Open this new project and Navigate to the placement view by navigating to the Device Editor and clicking on the Interconnect View icon.
- The jumpers remain the same.
- Regenerate the application.

When **Bit3** of the **CPU_SCR0** register is set high, the system is placed into a low power sleep mode. In this mode, the system clock is disabled. To wake up the system, this register bit must be cleared. The CPU cannot clear this bit, since it is not operating. It can only be cleared by the presence of any pending interrupt. Setting the appropriate bit of one of the **INT_MSKx** registers enables a pending interrupt. The sleep timer interrupt is enabled when **Bit6** of **INT_MSK0** is set high.

To provide more readable code, the system defines **M8C_Sleep** as **(CPU_SCR0 |= 0x08)**. It causes the system to enter the sleep mode.

Step 2. Modify the Code.

- Navigate to the Application Editor and open **main.c**.
- Make the following modifications.
  1. Using **INT_MSK0**, enable the sleep timer interrupt.
  2. Replace the section of code that waits for a posted interrupt with a sleep command.
- Rebuild the project, download to the Eval1 board, and run it.
- Verify that the program works correctly.

Exercise 2B-1: Why might you want to use the sleep mode instead of polling for the posted sleep timer interrupt?

Exercise 2B-2: Why might you want to poll instead of putting the system into sleep mode?

1.2.3 Lab 2C - Global Interrupts

Step 1. Create New Project.

- Make a copy of **Lab2B** and name it **Lab2C**.
- All parameters and jumper connection remain the same.
- Generate the application,

The posted interrupt can be "forced" either by clearing **INT_CLR0** bit 6, or by letting the interrupt be serviced. The smallest interrupt service routine (**ISR**) is created by placing a "**reti**" at the interrupt's vector address. All this routine does is return the CPU operation to its original program. This type of ISR is defined as a "**stub**".

- Open **boot.asm** and verify that there is a stub in the sleep timer vector location. 64h

Exercise 2C-1: How much code space is available at this vector location?

Exercise 2C-2: Suppose more space is needed, what can you do?

Setting Bit0 of the flag "F" register enables the global interrupt. Conversely clearing the same bit disables the global interrupt. For more readable code the system defines **M8C_EnableGInt** as **asm(or "F, 01h")**. It causes the global interrupt to be enabled. "**asm( )**" is a function that allows assembly commands to be embedded into a C program.

Step 2. Modify the Code.

- Navigate to the Application Editor and open **main.c**.
- Make the following modifications.
  1. Add code to enable the global interrupt.
  2. Remove the code used to clear the sleep timer interrupt.
- Rebuild the project, download it to the Eval1 board, and run it.
- Verify that the program works as before.

A code example (also available in Lab2C) is shown below:

```
//-------------------
// Lab2C Control Software
//-------------------
#include <m8c.h> // part specific constants and macros
#include "PSoCAPI.h" // PSoC API definitions for all User Modules
extern unsigned char bShadow = 0;
void main(){
PRT1DR = 0;
        INT_MSK0 |= 0x40;
        M8C_EnableGInt;
        while(1){
                M8C_Sleep;
                bShadow++;
                PRT1DR = bShadow;
            }
}
```

1.2.4 Lab 2D - Assembler Interrupt Service Routines (ISRs)

Step 1. Create a New Project.

- Make a copy of **Lab 2C** and name it **Lab 2D**.
- All the parameters and jumper connections remain the same.
- Generate the application

Step 2. Modify the code.

- Create a new Assembly Source file named **SleepTimerRoutines.asm**.
- Open this new file and add the code shown in Figure 1.16.

SleepTimerRoutines.asm

```
;----------------------
;// SleepTimer ISR
;----------------------
include  "m8c.inc"
export  SleepTimerISR

SleepTimerISR:
   push  A
   inc   [_bShadow]
   mov   A,[_bShadow]
   mov   reg[PRT1DR],A
   pop   A
   reti
```

Figure 1.16: SleepTimer ISR.

Note that:

- The accumulator value is stored by "pushing" it onto the stack and then later it is recovered from the stack. The interrupt hardware only preserves the program counter and flag register. These are the minimum requirements for an ISR. Other registers used by the main program may also need to be preserved. Generally this is done by temporarily storing them on the stack.
- **bShadow** is global variable that was declared in **main.c**. When used by assembly routines it must have an underscore in front of it. Conversely any global variable declared in an assembly file must start with an underscore for a C routine to be able to find it.
- The routine ends with a **reti**.
- Open **boot.asm** and place the following code at the sleep timer vector location.

ljmp SleepTimerISR

Exercise 2D-1: How many bytes are required for this instruction?

- Regenerate the project.
- Open **boot.asm** and verify that the code you just entered has disappeared. This is because **boot.asm** is rewritten every time the project is regenerated. For the code to remain, it much be entered into the template file used to generate boot.asm.
- Open **boot.tpl** and place the following code at the sleep timer vector location.

ljmp SleepTimerISR

Use File Open "All Files"

- Regenerate the project.
- Open **boot.asm** and verify this code now resides in the correct vector location.
- Open **main.c** and remove all code in the control loop. The program remains in this loop, while the sleep timer is serviced in its ISR.
- Rebuild the project, download it to the Eval1 board, and run it.
- Verify that the program works as before.

1.2.5 Lab 2E - C Language Interrupt Service Routines (ISRs)

Step 1. Create a New Project.

- Make a copy of **Lab 2D** and name it **Lab 2E**.
- All parameters and jumper connection remain the same.
- Generate the application.

Step 2. Modify the code.

- Remove **SleepTimerRoutines.asm**. Right click and "Exclude from Project
- Add the code to **main.c** shown in Figure 1.17.

main.c

```
//-------------------------------
// Lab2E Control Software
//-------------------------------
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules

extern unsigned char bShadow = 0;
void main(){
    PRT1DR = 0;
    INT_MSK0 |= 0x40;
    M8C_EnableGInt;
    while(1){
    }
}

#pragma interrupt_handler SleepTimerISR

void SleepTimerISR(void){



}
```

Figure 1.17: Lab 2E Control software.

Note that:

1. The **pragma**[3] allows a function to be used as an interrupt handler. The compiler will automatically add the code necessary to preserve any registers it uses. It will also add the **reti**.

[3] A pragma is code inserted into the source code that serves as special instructions to the compiler, but is otherwise ignored.

2. Of course, such a function can neither accept parameters, nor return a result.

Exercise 2E-1: Why?

- Add the code that is required to implement the previous assembly ISR.
- Open **boot.tpl**, navigate to the sleep timer vector location, and add an underscore to **SleepTimerISR**.

Note: For functions and variables declared in C, an underscore is added. If **iCVar** is a variable declared in C, then it is accessible by routines such as **_iCVar** and C routines as **iCVar**. If **_iAsmVar** is a variable declared in assembly, then it is accessible by assembly routines as **_iASmVar** and C routines as **iCVar**. **iAsmVar** is a variable declared in assembly, then it is accessible by assembly routines as **iASmVar** and not accessible by C routines.

- Regenerate the application, rebuild the project, download it to the Eval1 board, and run it.
- Verify that the program works as before.

1.2.6 Lab 2F - Forcing an Interrupt

Step 1. Create a New Project.

- Make a copy of **Lab2E** and name it **Lab2F**.
- All parameters and jumper connections remain the same.
- Generate the application.

There are times when it may be advantageous to force an interrupt via software control by setting the interrupt's **INT_CLRx** register bit, e.g., Bit6 of the **INT_CLR0** register for the sleep timer. To insure that this bit is not accidentally set, the enable software interrupt bit (**ENSWINT**) must be set. It is located at Bit7 of **INT_MSK3**. Table 1.4 shows the possible results for manipulations of an interrupt's **INT_CLRx** bit and **ENSWINT**.

Table 1.4: **INT_CLRx** and **ENSWINT** action/result table.

Action	Result
Read a zero from **INT_CLRx** bit.	No interrupt has been posted
Read a one from **INT_CLRx** bit.	Interrupt has been posted
Write a zero to **INT_CLRx** bit when **ENSWINT** is zero	Clear posted interrupt
Write a one to **INT_CLRx** bit when **ENSWINT** is zero	No action
Write a zero to **INT_CLRx** bit when **ENSWINT** is one.	No action
Write a one to **INT_CLRx** bit when **ENSWINT** is one.	Set a posted interrupt.

Exercise 2F-1: In previous Labs the posted interrupt was cleared without adding code to set ENSWINT to zero. Why did this work?

Step 2. Modify the code.

- Add code to enable ENSWINT.

- Add code to force the sleep timer to post an interrupt to the control loop.

Exercise 2F-2: How should this program work now?

- Regenerate the project, rebuild the project, download to the Eval1 board, and run it.

Exercise 2F-3: Does the actual operation agree with your prediction?

1.2.7 Lab 2G - Creating Very Small Interrupt Service Routines (ISRs)

Step 1. Create a New Project.

- Make a copy of **Lab 2F** and name it **Lab 2G**.
- Set the set the **Sleep_Timer** global parameter to **64_Hz**.
- All other parameters and jumper connections remain the same.

Sometimes an interrupt server is so small that it reside in the four byte space allocated for its input vector.

Step 2. Modify the Code.

- Remove the interrupt handler located in **main.c**.
- Create a global single byte variable named **bCount**.
- Remove the code used to enable **ENSWINT**.
- Remove the code used to force an interrupt.
- Open **boot.tpl** and add assembly code to the sleep timers interrupt vector location that decrements this new variable *(be sure to keep the reti.)*

Exercise 2G-1: How many bytes of code does this handler take?

- Add code in the control loop to:

1. Turn on all four LEDs.
2. Set **bCount** to 255.
3. Wait till **bCount** counts down to zero.
4. Turn Off LED4.
5. Set **bCount** to 192
6. Wait till **bCount** counts down to zero.
7. Turn Off LED3.
8. Set **bCount** to 128
9. Wait till **bCount** counts down to zero.
10. Turn Off LED2.
11. Set **bCount** to 64.
12. Wait till **bCount** counts down to zero.
13. Turn Off LED1.
14. Set bCount to 32

15. Wait till **bCount** counts down to zero.

Exercise 2G-2: How should the output behave?

- Regenerate the application, rebuild the project, download it to the Eval1 board, and run it.

Exercise 2G-3: Does actual operation agree with your prediction?

1.3 Lab 3 - Pulse Width Modulation

Pulse width modulation is a popular technique used for controlling analog devices by using digital techniques. It is widely used for motor control, communications, power control, measurement, etc. It is a digital technical because power is either full on, or full of,f but with a variable duty cycle. PSoC has both 8 and 16-bit pulse width modulators with programmable pulse width and period.

Objective: This Lab is an introduction to the Pulse Width Modulator User Module. Note that this Lab is divided into four parts, Lab3A through Lab3D, respectively.

Upon successful completion of this Lab you will:

- understand the operation a PWM,
- understand how to generate three variable clocks,
- understand how to connect a User Module Output to a pin via the global output resources,
- understand the routing limitations for User Module Outputs,
- understand the clock synchronizing requirements,
- have used the row embedded digital logic to combine the outputs of two PWMs to generate a more complex waveform,

and,

- have used the CPU to control the Pulse Width of a PWM.

Assumptions:

- All previous Lab assumptions.

Required Materials:

- CY3210 PSoCEval1 Board.
- Breadboard wire.

Required Equipment:

- Oscilloscope.

Related Reference Material:

- All previous Lab reference materials.

1.3.1 Lab 3A - Global Output

Digital Blocks connect to the pins via global interconnection rows and columns.

Step 1. Create a New Project.

- Name it **Lab3A**.
- Select the C option.
  (*Unless stated otherwise, all projects are to be written in C.*)
- Use the standard Lab settings for part type and settings, e.g., CPU Clock = 12 MHz and supply voltage = 5V.
- Navigate to the Interconnect View and set the parameters as shown in Table 1.6.

Table 1.6: Clock Parameter Settings.

Parameter	*Value*
Sleep Timer	*64_Hz*
VC1=SysClK/n	*1*
VC2 = VC1/n	*16*
VC3 Source	*SysClk*2*
VC3	*3*

Exercise 3A-1: What is the frequency of each variable clock?

VC1 = _____ VC2 = _____ VC3 = _____

- Set the four port pins as shown in Table 1.7.

Step 2. Add jumpers to the Eval1 board.

- Use jumper wires to make the following connections:
  1. P14 to LED1
  2. P00 to LED2

Table 1.7: Drive Settings

Name	Port	Select	Drive
PWMOut1	P1[4]	StdCPU	HighZAnalog
PWMOut2	P0[0]	StdCPU	HighZAnalog
PWMOut3	P0[4]	StdCPU	HighZAnalog
SwitchIn	P1[3]	StdCPU	PullDown

3. P04 to LED3
4. P13 to SW

This configuration is shown in Figure 1.19.

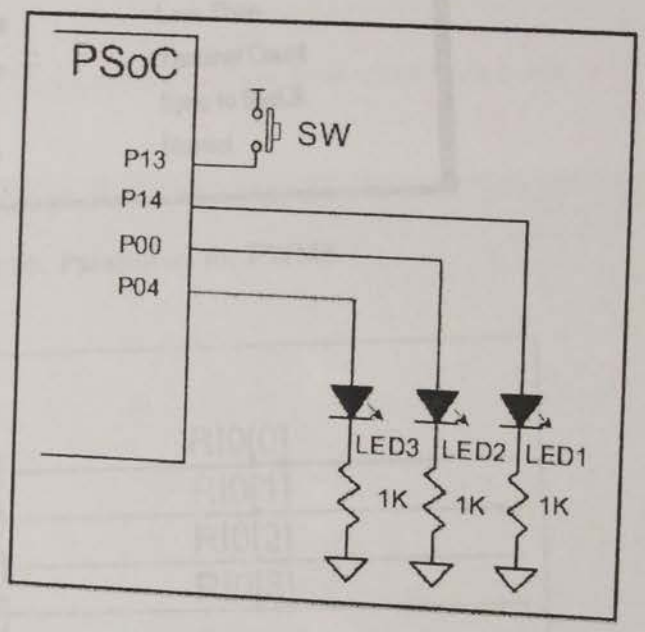

Figure 1.19: LED connections.

Step 3. Place and Parameterize a User Module.

- Select a **PWM8** User Module. It should be named **PWM8_1**.
- Rename it **PWM8** and place in **DBB00**.
- Set its parameters as shown Figure 1.20.

  Note the Period value is shown as 255. Because the counter decrements through zero there are 256 actual steps. The convention will be to define the required period in terms of the value that User Module parameter requires to set it. As an example, for a period of 100 steps, **Period** is set to 99. The placed block should appear as shown in Figure 1.21.

The clock is connected to VC2, the PWM is always enabled, and its compare out is connected to row 0 output 0 (**RO0[0]**).

Step 4. Connect PWM to the desired Pin.

PWM8

User Module Parameters	Value
Clock	VC2
Enable	High
CompareOut	Row_0_Output_0
TerminalCountOut	None
Period	255
PulseWidth	32
CompareType	Less Than
InterruptType	Terminal Count
ClockSync	Sync to SysClk
InvertEnable	Normal

Figure 1.20: Parameters for PWM8.

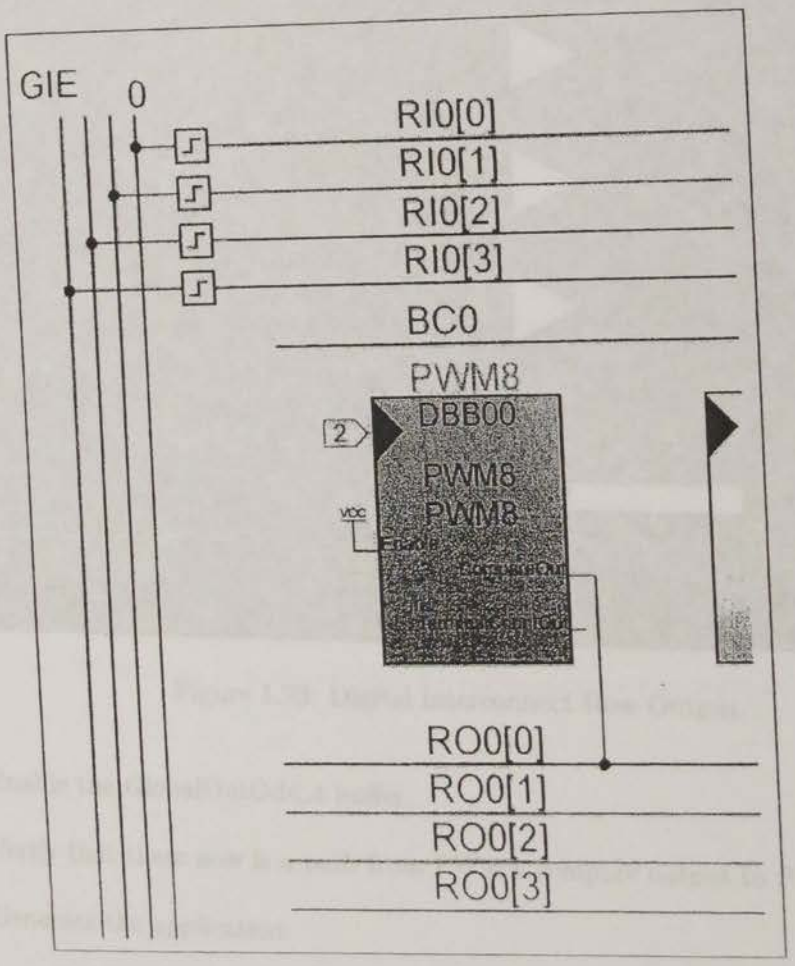

Figure 1.21: Interconnect View for PWM8.

- Select P1[4] to be GlobalOutOdd_4. Note that the drive changed to **Strong**.
- Verify that P1[4] is connected to the GlobalOutOdd_4 column.
- At the far right of RO0[0] is a box, as shown in Figure 1.22. Clicking on this box opens the window shown in Figure 1.23.

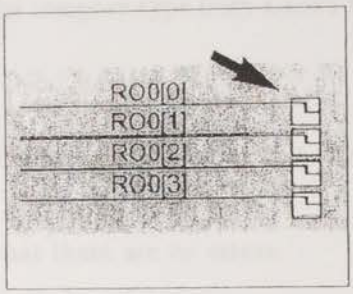

Figure 1.22: Location of Digital Interconnect click box.

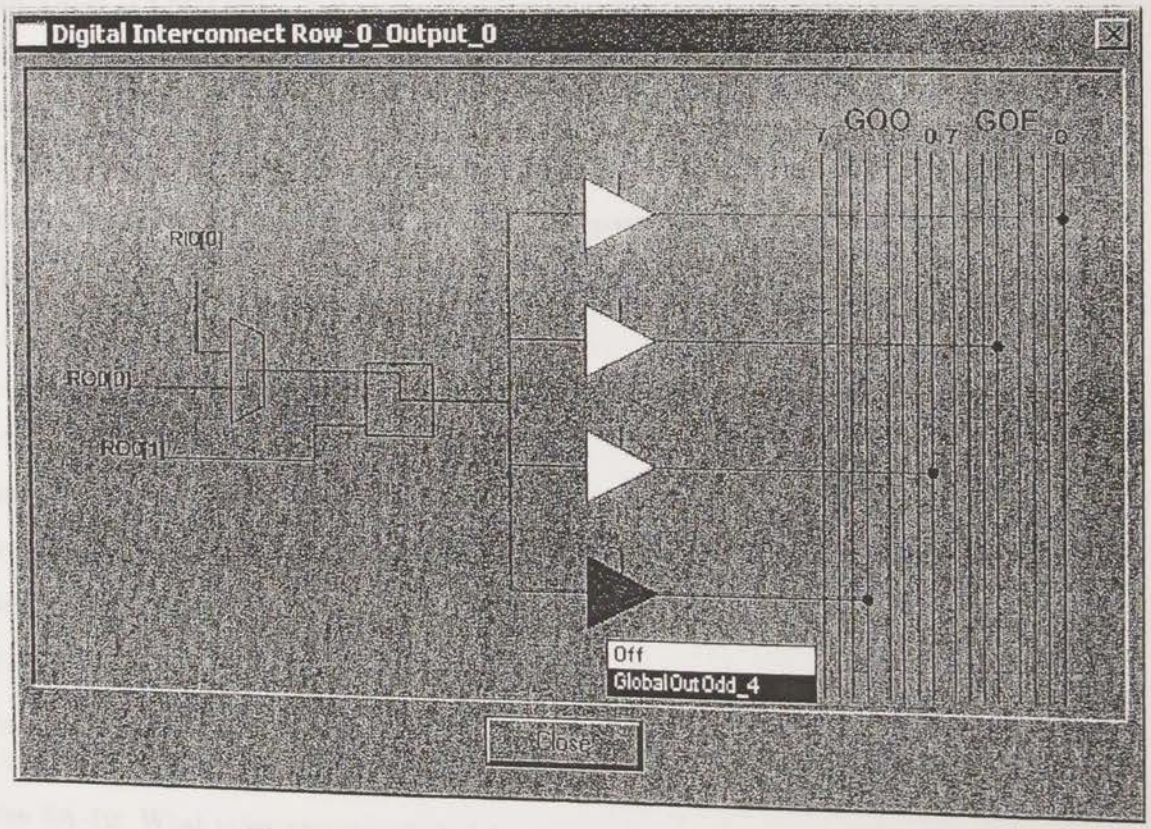

Figure 1.23: Digital Interconnect Row Output.

- Enable the GlobalOutOdd_4 buffer.
- Verify that there now is a path from PWM's compare output to P14.
- Generate the application.

Exercise 3A-2: Having set all these parameters, what is the output frequency of the PWM signal?

Exercise 3A-3: How would changing the compare type from **LessThan** to **LessThanOrEqual** change the waveform?

Step 4. Write the software.

- Open "**main.c**" and add the necessary code to the start the PWM User Module, and include an empty control loop.
- Build this application and verify that there are no errors.
- Download the program to the Eval1 board and run it.
- Verify correct operation by observing the LEDs.

Exercise 3A-4: Is LED1 on?

Using an oscilloscope, measure the frequency and pulse width of the PWMOut.

Exercise 3A-5: Are these reading consistent with your predictions?

Step 5. Place and parameterize another User Module.

- Select another PWM8 User Module and rename it **PWM8a**.
- Place it in **DBB10**.
- All of its parameters should be the same except for:
  - ☐ Connect its compare output to P00.
  - ☐ Set its PulseWidth parameter to 224

Exercise 3A-6: How should these two waveforms differ?

Exercise 3A-7: Does your observation confirm your prediction?

Exercise 3A-8: Is this difference apparent when viewing LEDs?

Step 6. Place and parameterize a Third User Module.

Exercise 3A-9: Either place and parameterize another PWM and connect it to P04, or explain why it is not possible.

Exercise 3A-10: What other pins are blocked for connection by this PWM?

1.3.3 Lab 3C - Global Output

Embedded digital logic in the output rows allows complex waveforms to be generated by multiple digital blocks.

Step 1. Create a New Project.

- Name it **Lab3C**.
- Select the C option. (*Unless stated differently all project will be written in C*).
- Use the standard Lab settings for part type and the settings. (12 MHz CPU Clock, 5V).
- Navigate to the Interconnect View and set the parameters as shown in Table 1.9.

Table 1.9: Parameter Settings

Parameter	*Value*
Sleep Timer	*64_Hz*
VC1=SysClK/n	*2*
VC2 = VC1/n	*16*
VC3 Source	*VC1*
VC3	*256*

Exercise 3C-1: What is the frequency of VC3?

Table 1.10: Port Drive Settings

Name	Port	Select	Drive
PWMOut	P1[4]	StdCPU	HighZAnalog
SwitchIn	P1[3]	StdCPU	PullDown

- Set the ~~four~~ two port pins as shown in Table 1.10.

Step 2. Add jumpers to the Eval1 board.

- Use jumper wires to make the following connections:
  1. P14 to LED1
  2. P13 to SW

This configuration is shown in Figure 1.25.

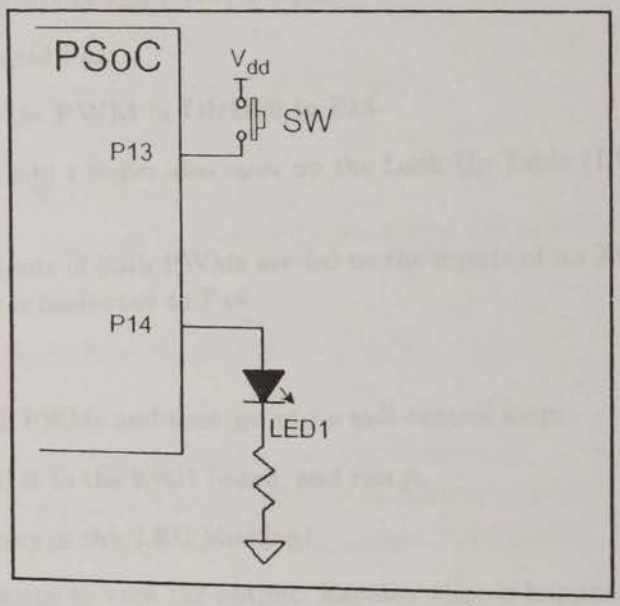

Figure 1.25: Global Output schematic.

Step 3. Place and Parameterize a PWM.

- Select and place a PWM8 User Module in **DBB00**.
- Set the following parameters:
  - ☐ The Clock is V3
  - ☐ It is always Enabled
  - ☐ The **Period** is 255
  - ☐ The **PulseWidth** is 128
  - ☐ The **CompareType** is LessThan

- ☐ The compare output connects to row 0 output 0.

Exercise 3C-2: What is the frequency of the PWM Output?

Step 4. Place and Parameterize a Second PWM.

- Select and place a PWM8 User Module in **DBB01**.
- Set the parameters so that:
  - ☐ The Clock is V3.
  - ☐ It is always Enabled.
  - ☐ The **Period** is 253.
  - ☐ The **PulseWidth** is 127.
  - ☐ The **CompareType** is LessThan.
  - ☐ The compare output connects to row 0 output 1.

Exercise 3C-3: What is the frequency of this PWM Output?

Step 5. Connect PWM to the desired Pin.

- Connect the compare out of the **PWM** in **DBB00** to P14.
- While setting the global out odd 4 buffer also open up the Look Up Table (**LUT**) and set it to **A_ XOR_B**.
- Verify that the compare outputs of both PWMs are fed to the inputs of an XOR gate and that the output of this gate is connected to P14.
- Generate the application.
- Write software to start both PWMs and then go into a null control loop.
- Build the project, download it to the Eval1 board, and run it.

Exercise 3C-4: At what frequency is this LED blinking?

Exercise 3C-5: Use the oscilloscope to view the output. Explain what is happening.

Exercise 3C-6: How must you change the clock so that it blinks at a one Hertz rate?

Exercise 3C-7: D so by changing only the VC1, VC2, and VC3 parameters.

1.3.4 Lab 3D - Modifying a PWM Pulse Width under Software Control

It is quite common for the CPU to manipulate the period, or pulse width, of a PWM as, for example, in motor controllers and battery chargers.

This Lab's control loop will:

- poll (and clear) the sleep timer interrupt,
- read and de-bounce the input switch,
- increment the pulse width value, if the switch has been pressed.

and,

- display the value on the LCD (Liquid Crystal Display) in decimal, hexadecimal, or other format, anytime the pulse width value has been changed.

Step 1. Make a copy of Lab3C.

- Rename it **Lab3D**.
- Go into the folder and rename **Lab3C.soc** to be **Lab3D.soc**.
- Delete the PWM located in **DBB01**.
- Change the LUT so that only the remaining PWM controls P14.
- Open this new project and navigate to the placement view.
- All global parameters remain the same.
- All pin parameters remain the same.
- Use parameters remain the same.
- All wire connections remain the same.
- Add a LCD Use Module. Assign it to Port 2.
- Generate the Application.

Step 2. Write the control software.

- Open "**main.c**" and add code the start the PWM and LCD User Modules.
- Initialize the LCD.
- Display the initial Pulse Width Value on LCD.
- In a control loop:
  - ☐ Wait for the sleep timer interrupt and clear it.
  - ☐ Read the input switch and perform de-bounce algorithm.
  - ☐ If switch has just been pressed, increment the Pulse Width value and update the display.
- Generate the application.

Here is a clever algorithm for de-bouncing a switch (The switch is located at P13):

```
bSwitchState <<= 1;
   bSwitchState &= 0x70;
   bSwitchState |= (PRT1DR & 0x08);
   if(bSwitchState == 0x38){
   //Switch has just been pressed
   // Do something
   }
```

Exercise 3D-1: Describe how this works.

- Build the application, download it to the Eval1, and run it.
- Verify that the LED get brighter as the pulse width increases.

1.4 Lab 4 - Three Wire Fan, Tachometers, Global Inputs.

Fan controllers are an important application of PSoC technology. Many fans include an integral tachometer which is used to monitor and control a fan's speed. This lab will utilize a pulse width modulator as a simple method of controlling a fan's speed. Since the measurement of a fan's speed is in fact a measurement of frequency, frequency measuring techniques are also discussed and a method is presented for frequency measurement using PSoC.

Objective: This Lab introduces the major theme of driving and controlling a three wire fan. Controlling a fan's speed requires a loop that:

1. Pulse Width Modulates a fan's power.
2. Measures its speed.
3. Changes, the PWM's duty cycle by some predefined algorithm.

However, this Lab will address the first two of these three items.

On completion of this Lab you will:

- understand the operation of a three wire fan,
- have developed a three digital block PWM based control system,
- have developed a single digital block solution implemented with real time control in the interrupt handlers,
- understand how to connect a pin to User Module Input via the global input resources,

and,

- have used a Timer16 User Module to implement a tachometer.

Assumptions:

- All previous Lab assumptions.

Required Materials:

- A CY3210 PSoCEval1 Board.
- Breadboard wire.
- A 2N2222, or equivalent, NPN transistor.
- A 619 Ohm resistor.
- A 2.00K Ohm resistor.
- An ADDA Three Wire Fan (AB0305HB-GA6).

Required Equipment:

- Oscilloscope.

Related Reference Material:

- All previous Lab reference materials.
- Timer16 User Module Data Sheet.
- Cypress AN2283 (Frequency Meter) Application Note.

Three wire fans are common in industry and typically include temperature management for hardware systems.

The color code for the three wire fan is:

1. Red for power
2. Black for ground
3. Blue Tachometer

The schematic shown in Figure 1.26 and is the most basic configuration.

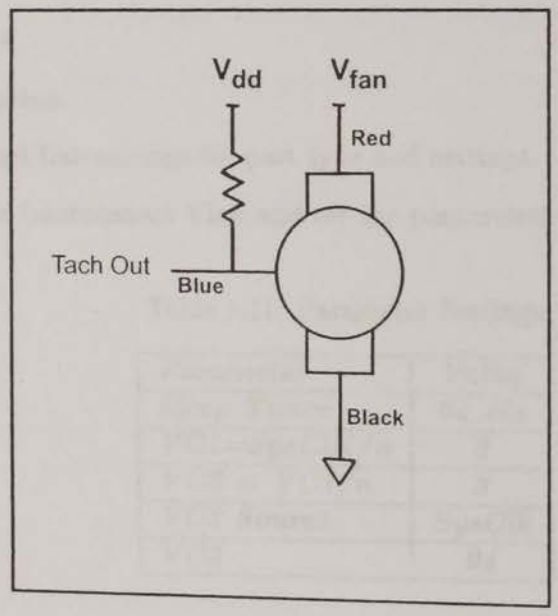

Figure 1.26: Typical motor/tachometer schematic.

A fan utilizes a motor whose speed is a function of the applied power. Originally tachometers were mechanical switches that were turned on and off by bumps on the motor drive shaft. Modern tachometers are electronic, but still only drive its output low. This allows for different voltage potentials for the fan and the control hardware. Note that the tachometer requires a pullup resistor.

Prior to beginning this Lab, connect the fan as follows:

1. Red lead to V_{dd}.
2. Black lead to ground.
3. Blue lead to one end of a 2.0K Ohm resistor.
4. Connect the other lead of the resistor to V_{dd}.

The fan blade should now be rotating. Use an oscilloscope to measure the frequency of the output signal output on the blue lead. Each fan has some number of poles. This particular fan has eight poles. For each revolution the tach sensor detects (N S N S N S N S). So for this fan, the tachometer generates a signal four times the fan speed. Note that it is necessary to know the number of poles a fan has before trying to measure its speed. Fan and motor speeds are expressed in terms of Revolutions Per Minute (**RPM**) and in this case is given by:

$$Speed_{RPM} = 60\left[\frac{TachFreq_{Hz}}{\frac{\#poles}{2}}\right] \tag{1.1}$$

Exercise 4-1: Using an oscilloscope to view the tachometer signal, what is the fan's speed in RPM?

1.4.1 Lab 4A - Driving a Fan with a PWM

Digital Blocks connect to output pins via global interconnection rows and columns. This allows versatility in selection of the particular pins used as outputs.

Step 1. Create New Project.

- Name it **Lab4A**.
- Select the **C** option.
- Use the standard Lab settings for part type and settings.
- Navigate to the Interconnect View and set the parameters as shown in Table 1.11.

Table 1.11: Parameter Settings.

Parameter	*Value*
Sleep Timer	*64_Hz*
VC1=SysClK/n	*8*
VC2 = VC1/n	*3*
VC3 Source	*SysClk*
VC3	*94*

Exercise 4A-1: What is the frequency of each variable clock?

VC1 = _____ VC2 = _____ VC3 = _____

Table 1.12: Port Drive Settings.

Name	Port	Select	Drive
SwitchIn	P1[0]	StdCPU	PullDown
TachIn	P1[2]	StdCPU	HighZ
FanPWMOut	P1[4]	GlobalOutOdd_4	Strong

- Set three port pins with the setting shown in Table 1.12.

Step 2. Build the Hardware.

- Configure the schematic as shown in Figure 1.27.

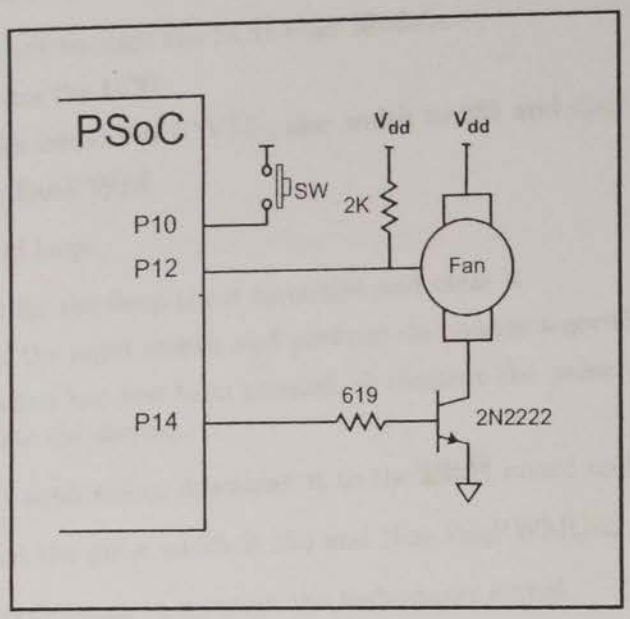

Figure 1.27: Fan schematic.

Step 3. Place and Parameterize a User Module.

- Select a PWM8 User Module, rename it **FanPWM** and place it in **DBB00**.
- It should:
  1. always be enabled,
  2. have a period of 255,
  3. have the **CompareType** parameter set to **Less Than** (*This allows a duty cycle from zero to 100%*),
  4. be clocked with **VC3**. This most closely sets the compare output frequency to **1000Hz**,
  5. have its compare output connected to **P14**.

  (Note that the pulse width is not important, since it will be set in software.)
- Select an LCD User Module, rename it **LCD**, and assign it to **Port_2**.
- Generate the application.

Exercise 4A-2: For this fan a PWM output frequency of 1000Hz is ideal. Suppose it had been 900Hz, what clock value would be required?

Exercise 4A-3: How would you generate it?

Step 4. Write the software.

- Open **main.c** and:
  1. Add code to start the LCD User Module.
  2. Initialize the LCD.
  3. Set the initial **FanPWM** pulse width to 255 and display it on the LCD.
  4. Start **FanPWM**
- In a control loop:
  1. Wait for the sleep timer interrupt and clear it.
  2. Read the input switch and perform de-bounce algorithm.
  3. If switch has just been pressed, decrement the pulse width value for **FanPWM** and update the display.
- Build this application, download it to the Eval1 board and run it.
- Verify that the pulse width is 255 and that FanPWMOut is on continuously.
- Use an oscilloscope to measure the tachometer signal.

Exercise 4A-4: What is the fan speed in RPM?

Exercise 4A-5: Lightly touch the fan blades with you finger. What happens to the frequency?

The tachometer only works when power is applied to the fan, i.e., when **FanPWMOut** is high.

- Press the button to change the pulse width to 254.
- Use one channel of an oscilloscope to view the tachometer signal and another to view the **FanPWMOut** signal.

Exercise 4A-6: Explain the mechanism that causes the tachometer waveform you see?

- Change the pulse width to 128.
- Use one channel of an oscilloscope to view the tachometer signal and another to view the **FanPWMOut** signal.

Exercise 4A-7: Viewing the tachometer output, does your previous explanation hold? If not develop another.

1.4.2 Lab 4B - Complex Motor Driver (Hardware Solution)

Lab 4A showed that the tachometer gives valid readings only when the fan is being driven. The standard solution is to periodically override the PWM and drive the fan "hard on" for some small window of time. During this window, the fan's frequency is measured. How small of a window and how often? It should be as small and as infrequent as possible. This is determined by the specific system requirements.

This particular fan controller must accurately control speeds from 2000 to 8000 RPM. This sets the lowest necessary frequency measurement to 133 Hz. One cycle of this frequency is 7.5 msec. A window of 15 msec assures that at least one cycle is captured. Adding 5 msec of design margin sets the window to 20 msec. The longer the time between samples, the less impact this window has on the control loop. For fan control applications a reasonable update rate is one per second.

So the PWM needs to be overridden for 20 msec, once per second, which is equivalent to a 2% override (20 msec/sec).

Step 1. Make a copy of Lab4A.

- Rename it **Lab4B**
- Set the four port pins with the setting shown in Table 1.13.

Table 1.13: Port Settings.

Name Set	Port	Select	Drive
SwitchIn	**P1[0]**	**StdCPU**	**PullDown**
TachIn	**P1[2]**	**StdCPU**	**HighZ**
FanPWMOut	**P1[4]**	**GlobalOutOdd_4**	**Strong**
TachOverrideOut	**P1[5]**	**GlobalOutOdd_5**	**Strong**

Step 2. Build the hardware.

- Configure the schematic as shown in Figure 1.28.

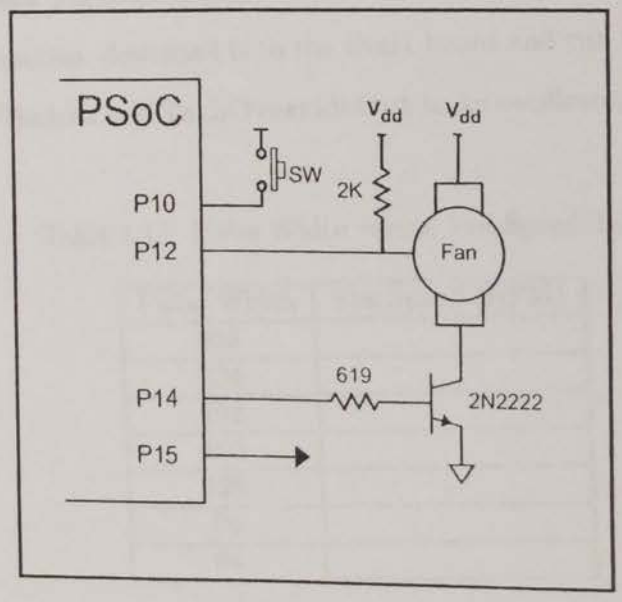

Figure 1.28: PSoC/Fan schematic.

Step 3. Place and parameterize a User Module.

- Select a PWM16 User Module, rename it **TachOverridePWM** and place it in **DBB01** and **DCB02**.
- It should:
  1. always be enabled,
  2. have a period of 1000,

3. have a pulse width of 20,
4. be clocked with the terminal count (TC) of **FanPWM**,
5. Have its compare output connected to **P15**.

- Set the LUT for **FanPWM**'s compare output to **A_OR_B**. This combines the override pulse with the PWM output signal.

The block diagram is shown in Figure 1.29.

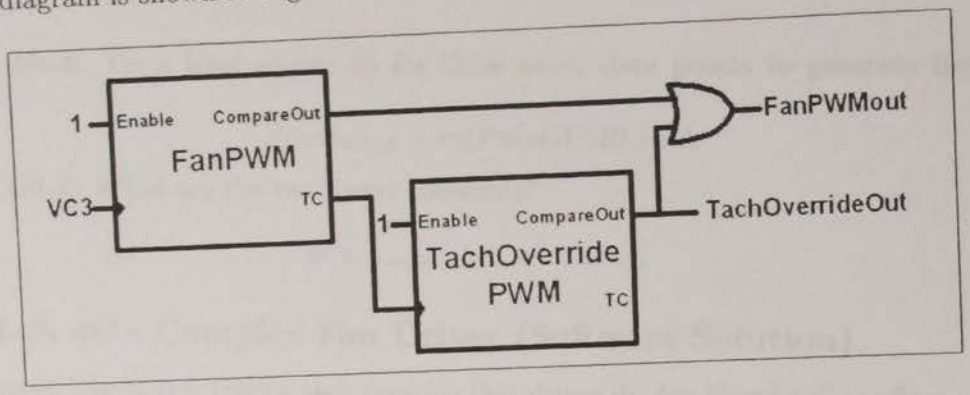

Figure 1.29: Block diagram for fan controller.

- Generate the application.

Exercise 4B-1: What is the period, and pulse width, of **TachOverrideOut**?

Exercise 4B-2: The **TC** output of **FanPWM** is used to clock **TachOverrideOut**. Why not use **CompareOut**?

Step 4. Write the software.

- Open **main.c** and add code to start **TachOverridePWM**.
- Build this application, download it to the Eval1 board and run it.
- Connect both **TachIn** and **TachOverrideOut** to an oscilloscope.

Table 1.14: Pulse Width versus Fan Speed Table.

Pulse Width	Fan Speed (RPM)
255	
224	
192	
160	
128	
96	
64	

Exercise 4B-3: Triggering the oscilloscope on the rising edge of **TachOverrrideOut**, what is the fan speed for the PWM pulse width values shown in Table 1.14 ?

Exercise 4B-4: Do a least square fit for these seven data points to generate the following equation.

$$Speed_{RPM} = m(PulseWidth) + b$$

Exercise 4B-5: What are the two linear constants?

$$m = \_\_\_\_\_ \quad b = \_\_\_\_\_$$

1.4.3 Lab 4C - Complex Fan Driver (Software Solution)

Lab 4B showed how to construct a tach override that drives the fan "hard on" to allow a frequency measurement. The previous solution required three of the eight digital blocks, or 37.5% of the digital block resources. If digital blocks are abundant, this may be acceptable. However, real world solutions rely heavily on the available digital resources and very quickly digital blocks become a scarce resource. Lab 4B was a hardware intensive solution and this Lab employs a state machine, as a software solution, placed in the FanPWM interrupt handler to generate the override signal.

Step 1. Make a copy of Lab 4A.

- Rename it **Lab 4C.**
- Set the four port pins with the setting shown in Table 1.15.

Table 1.15: Fan Controller Drive Settings

Name Set	Port	Select	Drive
SwitchIn	P1[0]	StdCPU	PullDown
TachIn	P1[2]	StdCPU	HighZ
FanPWMOut	P1[4]	GlobalOutOdd_4	Strong
TachOverrideOut	P1[6]	StdCPU	Strong

Step 2. Build the hardware.

- Configure the schematic as shown in Figure 1.30.

Step 3. Parameterize the FanPWM User Module.

- All parameters remain the same except that **InterruptType** is set to **TerminalCount.**
- Generate the application.

Step 4. Write the software.

- Open **main.c**
- Add code to:

1. Define a global 16 bit unsigned integer variable named **wControlState.**
2. Set its value to 1000.
3. Enable FanPWM's interrupt.
4. Enable the global interrupt.

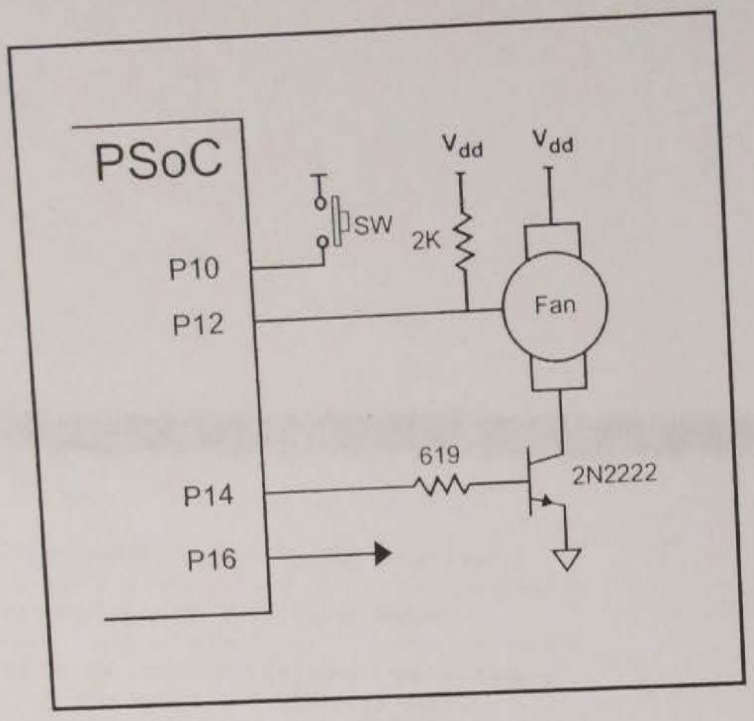

Figure 1.30: Fan controller schematic.

- Add the code shown in Figure 1.31 to the **FanPWM**'s interrupt handler (**FanPWMINT.asm**).

Note that:

1. Unlike the **SleepTimer** in Lab 1, user modules generate their own ISR handlers. They are automatically added to the vector table. Therefore it is not necessary to modify **boot.tpl**.

2. Within the interrupts handler, there is a space where user code may be placed. Note that if it is placed elsewhere, it is overwritten when the application is regenerated.

3. **wControlState** is a 2 byte variable. In assembly language code, the upper byte is available as [**_wControlState**] and the lower byte [**_wControlState + 1**].

4. The first two instructions are a clever way to decrement a two-byte variable.

5. The global output **FanPWMout** is forced high by setting all four bits of its LUT high (**RDIxLT0[3:0]**). The PSoC Technical Reference manual shows its address to be **0xb3**. (Setting the four bits of the LUT to 0x3 restores it to a single connection to the user module.)

- Build this application, download the program to the Eval1 board, and run it.

- Using an oscilloscope verify that **TachOverrideOut** is being generated.

- Verify that when this gate is high, **FanPWMOut** is forced high.

Exercise 4C-1: Describe another way to force **FanPWMOut** high?

fanpwmint.asm

```
_FanPWM_ISR:

   ;@PSoC_UserCode_BODY@ (Do not change this line.)
   ;---------------------------------------------------
   ; Insert your custom code below this banner
   ;---------------------------------------------------
   ;   NOTE: interrupt service routines must preserve
   ;   the values of the A and X CPU registers.
RDIxLTO:  equ  0xb3                    ;LUT Register Address
   dec   [_wControlState + 1]          ;decrement wContontrolState
   sbb   [_wControlState],0
   jnz   MoreThan255
   cmp   [_wControlState+1],20
   jnz   Not20
State20:
   mov   reg[PRT1DR], 0x40              ;Start of FanOverride
   or    reg[RDIxLTO],0x0f;             ;Set FanOverrideOut high
   reti                                ;force LUT open high
Not20:
   cmp   [_wControlState+1],0
   jnz   NotZero
State0:
   mov   reg[PRT1DR], 0x00              ;End of FanOverride
   and   reg[RDIxLTO],0xf3;             ;Set FanOverrideOut low
   mov   [_wControlState],3             ;force LUT back to normal
   mov   [_wControlState+1],232         ;wControlState = 1000
   reti
NotZero:
MoreThan255:
   ;---------------------------------------------------
   ; Insert your custom code above this banner
   ;---------------------------------------------------
   ;@PSoC_UserCode_END@ (Do not change this line.)

   reti
```

Figure 1.31: Fan PWM Interrupt Service Routine