PSoC® 4 Intelligent Fan Controller

Author: Rajiv Badiger
Associated Project: Yes
Associated Part Family: All 4200 parts
Software Version: PSoC Creator™ v4.0 or Higher

AN89346 demonstrates how to quickly and easily develop a four-wire brushless DC fan control system using PSoC® 4. The Fan Controller Component, available in PSoC Creator™, helps to manage the fans in a variety of configurations. This application note also shows how to combine fan control and temperature sensing to create a complete thermal management solution using PSoC 4.

Contents

1 Introduction

System cooling is a critical component of any high-power electronic system. As circuits become smaller, increasing demands are placed on system designers to improve the efficiency of system thermal management. This requires the system cooling fans to run at the optimum speed to ensure that the system temperature is always below a defined limit. To achieve this, a temperature measurement unit and a closed-loop fan speed controller are needed.

PSoC® 4 has the resources required to implement in a single chip a complete thermal management solution:

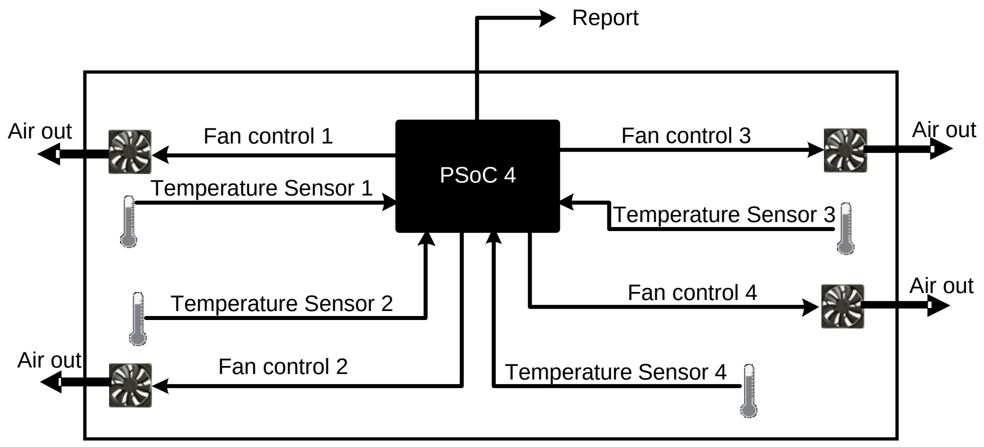
- The analog blocks, which include a SAR ADC, opamps, and IDACs, can be used to interface with analog temperature sensors such as a diode or a thermistor.
- The dedicated PWM hardware blocks can be used to drive as many as four independent fans. The universal digital blocks (UDBs) can be used for fan speed measurement using the tachometer signals from the fans.
- The powerful 32-bit ARM Cortex-M0 CPU can be used to implement firmware algorithms for measuring analog temperature sensors, and to implement the overall thermal management algorithms.

- The serial communication blocks (SCBs) can be used to report the fan status to an external host through the I^{2}C, SPI, or UART interface.

Figure 1 shows a block diagram of a PSoC 4 thermal management application.

PSoC 4 includes many other parts, beside PSoC 4200. They may have more or less on-chip PWM and UDB resources, which lead to slight difference in capacities on fan controller implementation. In this AN, the mentioned PSoC 4 is PSoC 4200 parts, if not specified otherwise.

Figure 1. PSoC 4 Thermal Management Application



PSoC Creator™, the integrated development environment (IDE) for PSoC, provides a FanController Component to simplify the design process. The component takes care of the closed-loop speed control. All you need to do is specify the required fan speed based on the system temperature. This application note shows how to use this component to build a PSoC 4 based thermal management system, with the help of three example projects:

- 'Project1_OpenLoop' – This project demonstrates the open-loop control mode where the firmware controls the nominal fan speed by setting the duty cycle of a PWM signal driving the fan.
- 'Project2_ClosedLoop' – This project shows how to implement a closed-loop fan-speed control system. The FanController Component automatically adjusts the PWM duty cycle to achieve a given fan speed.
- 'Project3_ThermalManagement' - The third example demonstrates a complete thermal management system that reads the temperature from external sensors and controls the fan speed to limit the temperature.

Other devices in the PSoC series – PSoC 1, PSoC 3, and PSoC 5LP – can also be used for a fan controller solution. See the following application notes for these devices:

- AN78692, PSoC 1 Intelligent Fan Controller
- AN66627, PSoC 3 and PSoC 5LP Intelligent Fan Controller

Select a PSoC device family based on the number of fans to be controlled and the number of temperature sensors to process. Table 1 shows a comparison between the PSoC families.

Table 1. Comparison of Different PSoC Families

Parameter	PSoC 1	PSoC 4	PSoC 3 and PSoC 5LP
Number of Fans	2	4	16
Sensor Types	RTD, Thermistor	RTD, Thermistor	Diode, RTD, Thermistor
I^{2}CSensors	Yes	Yes	Yes

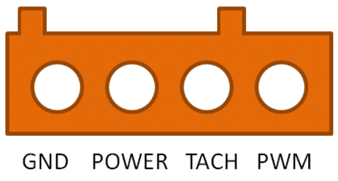
2 Four-Wire Fan Basics

Figure 2 shows a typical four-wire fan. Two of the wires are used to supply power to the fan. The other two wires are used for PWM and tachometer feedback for speed control and speed sensing respectively.

Figure 2. Typical 4-Wire DC Fan

At the cabling level, wire color-coding is not consistent across manufacturers, but the connector pin assignment is standardized. Figure 3 shows the connector pinout when viewed looking into the connector with the cable behind. Note that the connectors are keyed to prevent incorrect insertion into a fan controller board.

Figure 3. Four-Wire DC Fan Connector Pin Assignment



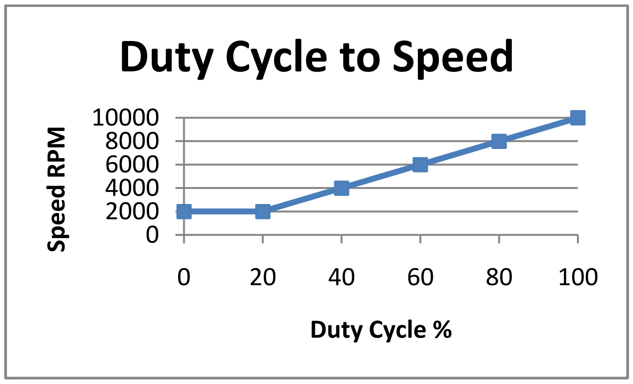
Fans come in standard sizes, with the common diameters being 40 mm, 80 mm, and 120 mm. One of the specifications to be checked when selecting a fan for a cooling application is how much air the fan can move. This is specified either as cubic feet per minute (CFM) or meters cubed per minute (m^3/min). The size, shape, and pitch of the fan blades all contribute to the fan's capability to move air.

Smaller fans are typically used in space-constrained applications. They must run at a higher speed to move the same volume of air in a given period; therefore, use more power and generate significantly more acoustic noise. This unavoidable tradeoff must be made to meet system requirements.

3 Fan Speed Control

The speed of a four-wire fan is controlled using a pulse-width modulator (PWM) signal. Increasing the duty cycle of the PWM signal increases the fan speed, as Figure 4 shows. Fan manufacturers specify how the PWM duty cycle relates to nominal fan speed using either a table of data points or a graph.

Figure 4. Example Duty Cycle to Speed Chart



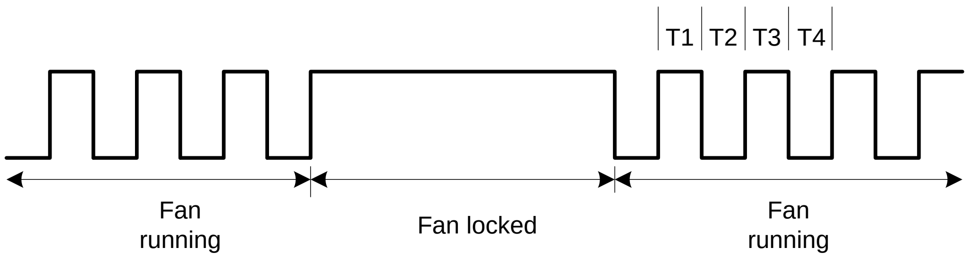
It is important to note that at low duty cycle, not all fans behave the same way. Some fans stop rotating as the duty cycle approaches 0%, while others rotate at a nominal specified minimum RPM. In both cases, the duty cycle to RPM relationship can be non-linear or not specified. A fan controller solution should include a mechanism to handle this fan behavior.

4 Measuring Fan Speed

DC fans include hall-effect sensors that sense the rotating magnetic fields generated by the fan's rotor as it spins. The output of the hall-effect sensor is a pulse train (referred as the tachometer signal) that has a frequency directly proportional to the fan speed. The number of pulses produced during each revolution depends on the number of poles in the electromechanical construction of the fan.

For the most common four-pole brushless DC fan, the tachometer output from the hall-effect sensor generates two high (T1 and T3) and two low (T2 and T4) pulses for each fan revolution, as Figure 5 shows.

Figure 5. Tachometer Output of a Fan



If the fan stops rotating due to mechanical failure or other fault, the tachometer output signal remains static at either a logic low or logic high level.

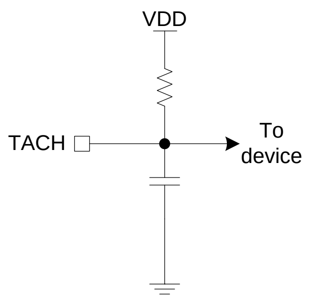
There are two ways to measure the rotation speed of a fan:

1. Use a high-frequency clock to measure the period of the tachometer pulse.
2. Count the tachometer pulses over a fixed period of time.

The selection of a particular method depends on the frequency to be measured. For a low-frequency input signal, it is better to use the first method as it provides a precise result with the measurement time dependent on the period of the input signal. For a high-frequency input signal, the second method is preferred, as the first method requires a very high frequency clock to achieve the same precision. In fan control applications, the tachometer signal frequency is around 600 Hz at 10,000 RPM for a four-pole fan. For such a frequency range, the first method can be used.

Since the tachometer line of a fan is an open drain / open collector output, it is necessary to pull the line high through a resistor, as Figure 6 shows. The input pins on PSoC 4 allow the use of built-in pull-up resistors so that an external resistor is not necessary. To avoid the effect of noise on speed measurements, put a small value capacitor of the order of a few nF to ground.

Figure 6. Tachometer Line Interface



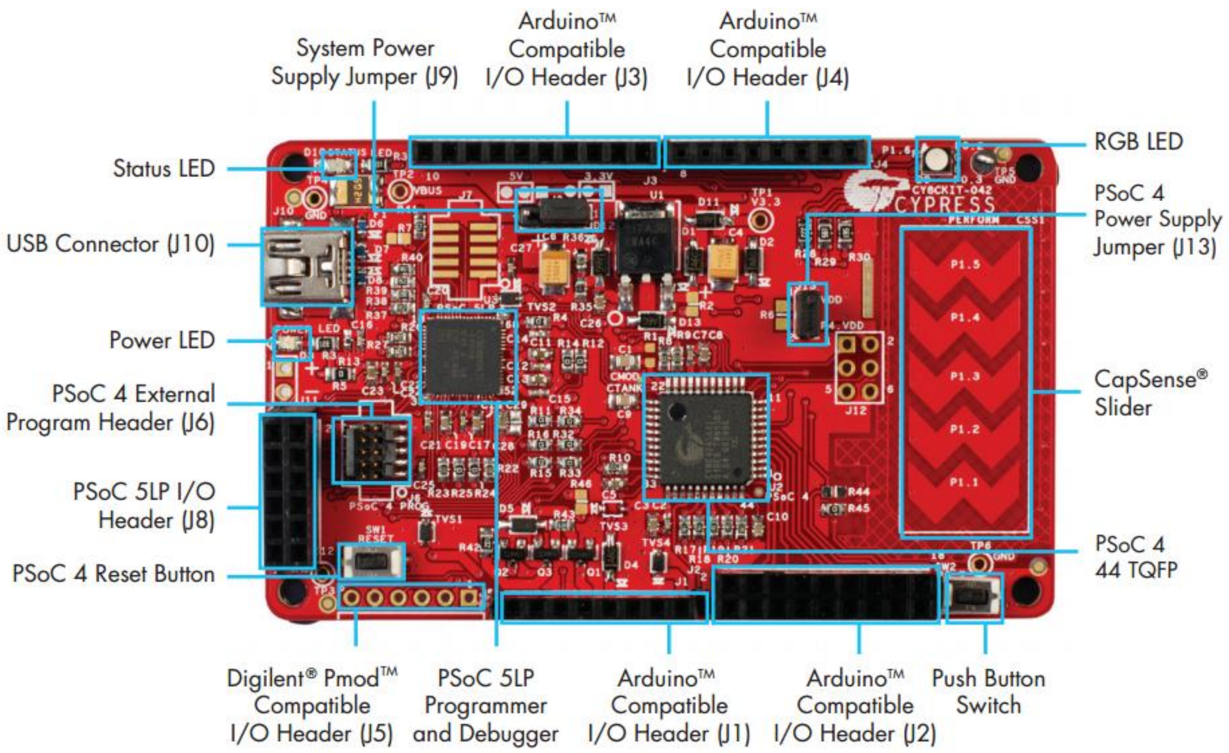
5 Cypress Development Kits for Thermal Management

The following Cypress kits can be used for quick and easy evaluations of thermal management solutions:

5.1 CY8CKIT-042 – PSoC 4 Pioneer Kit

This is a development board for the PSoC 4 device (Figure 7). It has an onboard programmer that is based on a PSoC 5LP device; it also acts as an I^{2}C-USB bridge. The board has Arduino-compatible headers, which you can use to connect third-party expansion boards.

Figure 7. CY8CKIT-042 PSoC 4 Pioneer Kit

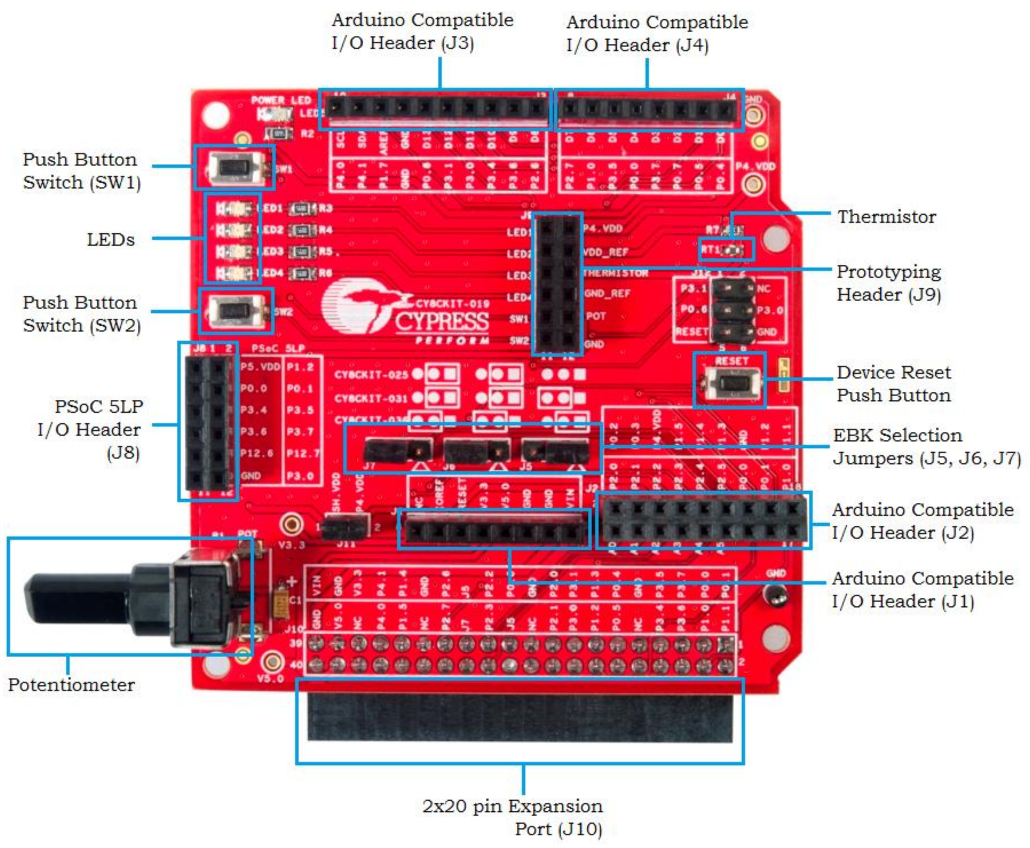

System Power Supply Jumper (J9)
Arduino™ Compatible I/O Header (J3)
Arduino™ Compatible I/O Header (J4)
RGB LED
Status LED
PSoC 4 Power Supply Jumper (J13)
USB Connector (J10)
Power LED
CapSense® Slider
PSoC 4 External Program Header (J6)
PSoC 5LP I/O Header (J8)
PSoC 4 44 TQFP
PSoC 4 Reset Button
Digilent® Pmod™ Compatible I/O Header (J5)
PSoC 5LP Programmer and Debugger
Arduino™ Compatible I/O Header (J1)
Arduino™ Compatible I/O Header (J2)
Push Button Switch

You can get more details about this kit at http://www.cypress.com/go/CY8CKIT-042.

5.2 CY8CKIT-019 – PSoC Shield Adapter

This adapter (Figure 8) enables you to connect a Cypress expansion board kit (EBK) to the PSoC 4 Pioneer kit.

Figure 8. CY8CKIT-019 PSoC Shield Adapter


Arduino Compatible I/O Header (J3)
Arduino Compatible I/O Header (J4)
Push Button Switch (SW1)
LEDs
Push Button Switch (SW2)
PSoC 5LP I/O Header (J8)
Potentiometer
Thermistor
Prototyping Header (J9)
Device Reset Push Button
EBK Selection Jumpers (J5, J6, J7)
Arduino Compatible I/O Header (J2)
Arduino Compatible I/O Header (J1)
2x20 pin Expansion Port (J10)

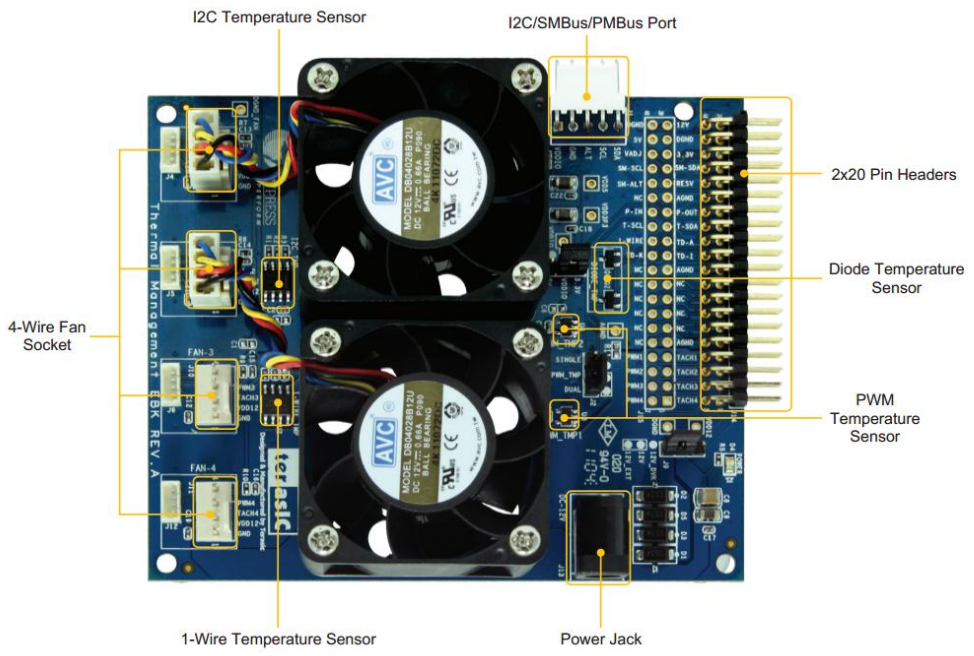
This kit has Arduino-compatible headers along with four LEDs, one potentiometer, and two switches.

You can get more details about this kit at http://www.cypress.com/go/CY8CKIT-019.

5.3 CY8CKIT-036 – PSoC Thermal Management Expansion Board Kit (EBK)

This is an EBK for a thermal management solution (Figure 9). It is used with a PSoC development kit. It comes with two fans plus a variety of analog and digital temperature sensors to enable you to quickly prototype a complete thermal management system.

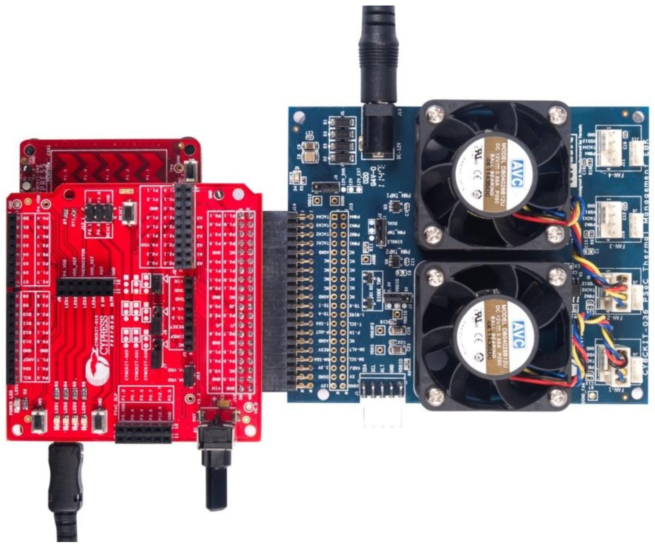
Figure 9. CY8CKIT-036 PSoC Thermal Management EBK



More information on the kit can be found at http://www.cypress.com/go/CY8CKIT-036.

You must have all the three kits to create a PSoC 4 based thermal management solution. Figure 10 shows the complete assembly.

Figure 10. Thermal Management Setup



The three example projects provided with this application note, and described in the following sections, can be easily tested on these kits.

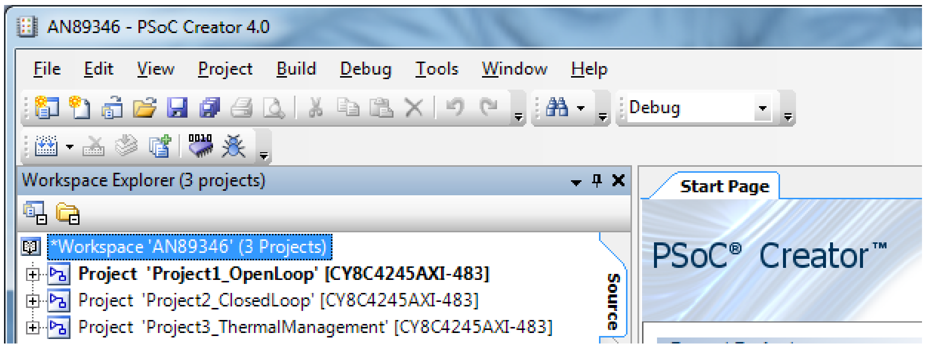
5.4 Example Projects

In the application note webpage, a *.zip* file of the projects is provided. Save and unzip it to your computer. Make sure that you have installed the latest PSoC Creator software, which can be downloaded here: http://www.cypress.com/psoccreator/

Double-click the PSoC Creator workspace file *AN89346.cywrk* to open the projects. In the Workspace Explorer window, three projects are listed, as Figure 11 shows:

1. Project1_OpenLoop
2. Project2_ClosedLoop
3. Project3_ThermalManagement

Figure 11. Workspace Explorer Window


AN89346 - PSoC Creator 4.0

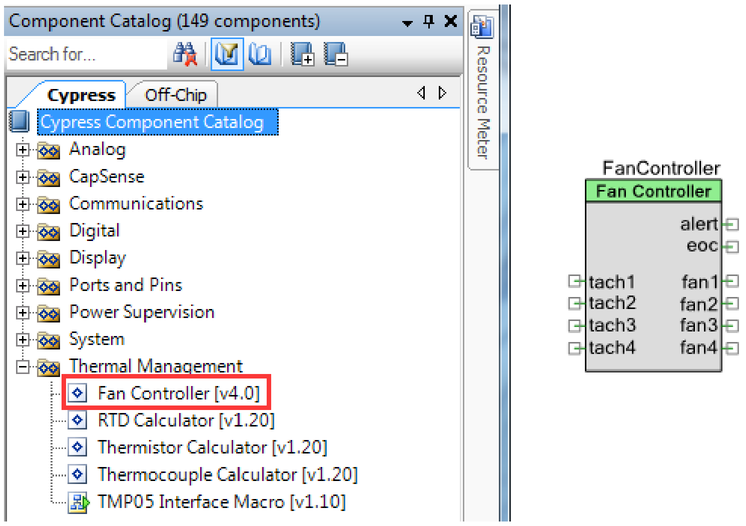
Workspace Explorer (3 projects)

'Workspace 'AN89346' (3 Projects)
Project 'Project1_OpenLoop' [CY8C4245AXI-483]
Project 'Project2_ClosedLoop' [CY8C4245AXI-483]
Project 'Project3_ThermalManagement' [CY8C4245AXI-483]

6 Project1_OpenLoop

This project demonstrates how to drive two fans in the open loop mode using the PSoC Creator FanController Component. Fan speed is set using the duty cycle value provided by an I^{2}C master, or by using the switches connected to the device. The FanController Component can be found under **Thermal Management** in the PSoC Creator **Component Catalog**, as Figure 12 shows.

Figure 12. FanController Component

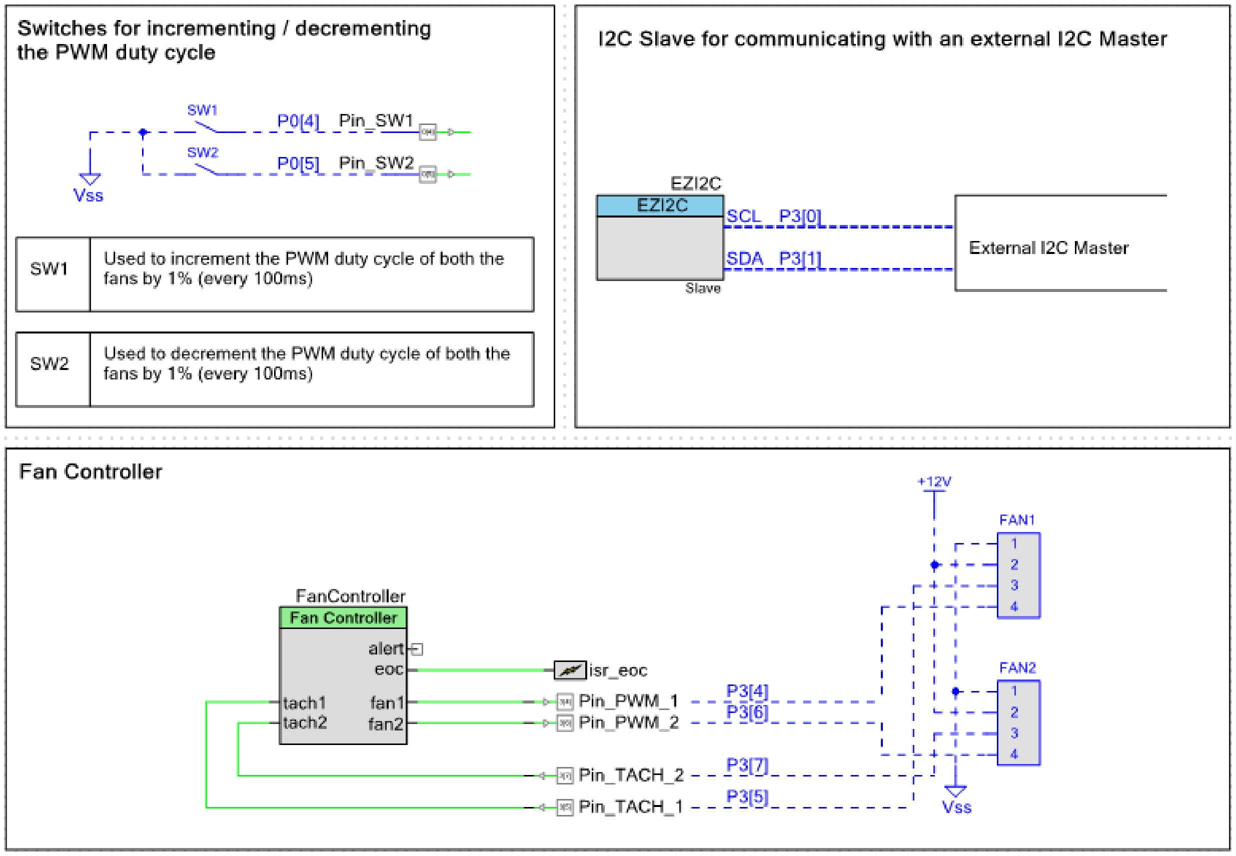

Component Catalog (149 components)

Cypress Component Catalog
- Analog
- CapSense
- Communications
- Digital
- Display
- Ports and Pins
- Power Supervision
- System
- Thermal Management
  - Fan Controller [v4.0]
  - RTD Calculator [v1.20]
  - Thermistor Calculator [v1.20]
  - Thermocouple Calculator [v1.20]
  - TMP05 Interface Macro [v1.10]

6.1 FanController Component

In the Workspace Explorer window (Figure 11 on page 8), expand the files under Project1_OpenLoop by clicking the '**+**' button next to the project. Click **Top Design** to open the project schematic. Figure 13 shows the Top Design schematic.

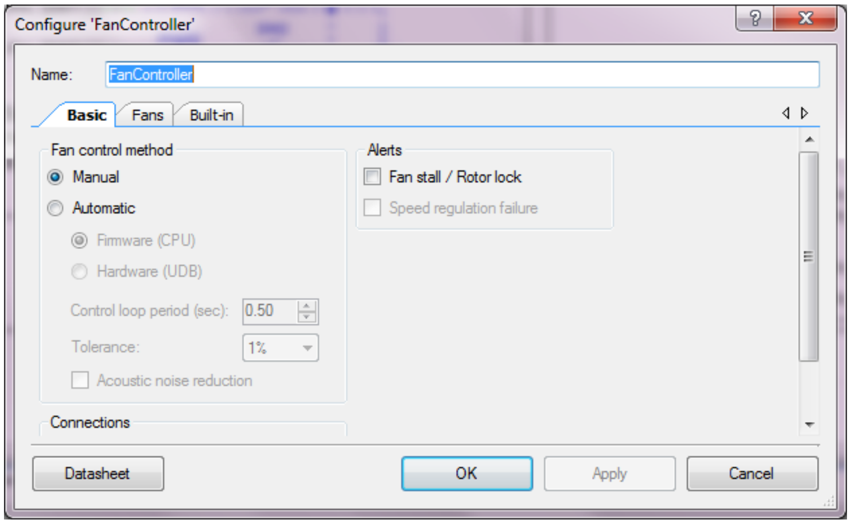
Figure 13. Project1_OpenLoop – TopDesign



SW1	Used to increment the PWM duty cycle of both the fans by 1% (every 100ms)
SW2	Used to decrement the PWM duty cycle of both the fans by 1% (every 100ms)

Double-click the FanController Component to open the Component Configuration Tool, as Figure 14 shows.

Figure 14. FanController Component Configuration Tool – Basic Tab



Name: FanController

Fan control method
- [✓ Manual]
- Automatic
  - [✓ Firmware (CPU)]
  - Hardware (UDB)
  - Control loop period (sec): 0.50
  - Tolerance: 1%
  - ☐ Acoustic noise reduction

Alerts
- ☐ Fan stall / Rotor lock
- ☐ Speed regulation failure

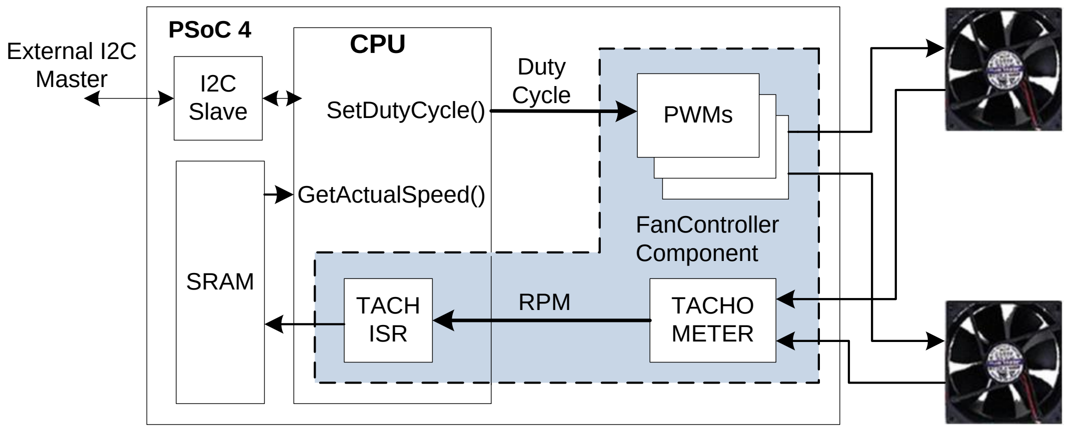
Connections

For this example project, the FanController Component is configured to use the **Manual** mode. In this mode, PSoC hardware blocks are used for the PWMs and the tachometer speed monitor, but firmware is responsible for setting the fan speed.

Figure 15 shows a simplified representation of the Manual control mode. In this mode, the API function SetDutyCycle() is used to set the individual PWM duty cycles, which in turn controls individual fan speeds. Speed measured by the Tachometer block is transferred to SRAM using an interrupt service routine. Fan speeds can then be read using the GetActualSpeed() API function. Note that this project does not use the actual fan speeds for any purpose.

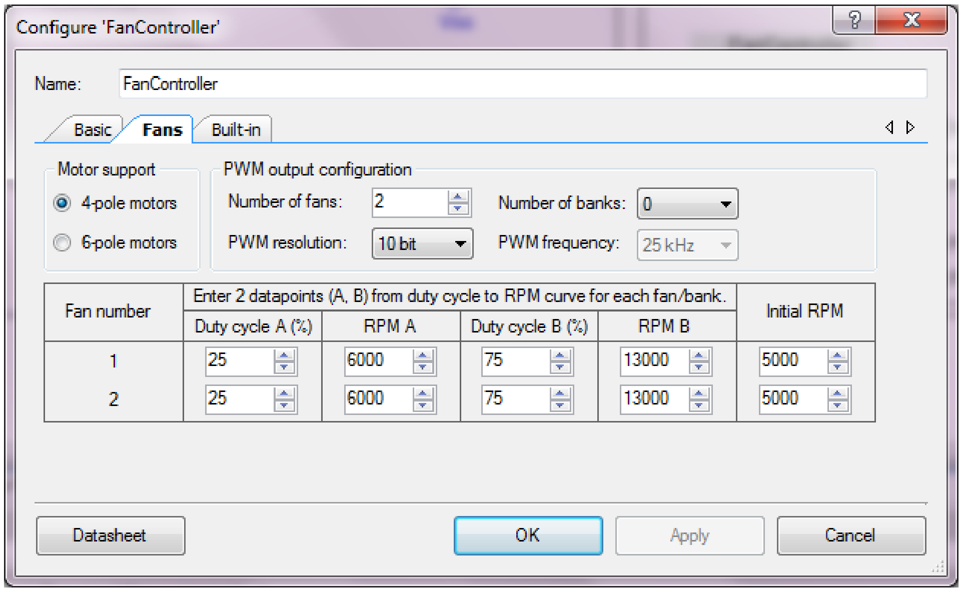
You can also configure the component to generate an alert signal if any fan stalls. Enable this feature by selecting the **Fan Stall / Rotor Lock** checkbox on the customizer **Basic** tab, but leave that box unchecked for the first example project.

Figure 15. Manual Control Mode – Block Diagram



Click the **Fans** Tab of the Configuration Tool to complete the configuration of the FanController Component, as Figure 16 shows.

Figure 16. FanController Component Configuration Tool – Fans Tab



Name: FanController

Motor support: [✓ 4-pole motors] 6-pole motors

PWM output configuration
Number of fans: 2
Number of banks: 0
PWM resolution: 10 bit
PWM frequency: 25 kHz

Fan number	Enter 2 datapoints (A, B) from duty cycle to RPM curve for each fan/bank.				Initial RPM
	Duty cycle A (%)	RPM A	Duty cycle B (%)	RPM B	
1	25	6000	75	13000	5000
2	25	6000	75	13000	5000

This tab enables you to configure all of the parameters related to the PWMs and the fans. Enter the number of fans and the fan bank arrangement. A fan bank is defined as multiple fans that are driven by the same PWM speed control signal. In this project, we drive two fans without banking.

The resolution of the PWM drivers can be set to 8-bit or 10-bit as required for your application. Ten-bit resolution gives finer speed control, but uses more digital resources. When the 8-bit resolution is selected, the PWM frequency can be set to either 25 kHz or 50 kHz. 25 kHz is commonly used for PWM drive signals, but fans that require higher PWM control frequencies can be supported with this option.

The lower portion of this tab enables you to enter the electromechanical parameters of your fans. If you have this information from the fan manufacturer's datasheet, enter it here. The **Duty cycle A (%)**, **RPM A**, **Duty cycle B (%)**, and **RPM B** parameters represent two data points from the fan's duty cycle -to- speed chart. If you are not able to get this information, keep the default settings for now. You will be able to measure the actual duty cycle to speed relationship for your fans in this project, and you can come back later and enter the measured parameters.

If you change the FanController configuration, rebuild the project, using menu **Build** > **Build Project1_OpenLoop**.

6.2 Other Project Components

After the FanController Component configuration is complete, note the other components placed in the design. The Digital Output Pin Components are used in the design to route the PWM signals to physical pins. The Digital Input Components with internal resistive pull-up are used to route the tachometer signals from the fans to the FanController Component.

An ISR Component is connected to the end-of-cycle (**eoc**) output of the FanController. The **eoc** signal is generated inside the component and pulses high when a new speed sample is available from all fans connected to the component. The resulting interrupt enables the CPU to read the fan speed at appropriate times.

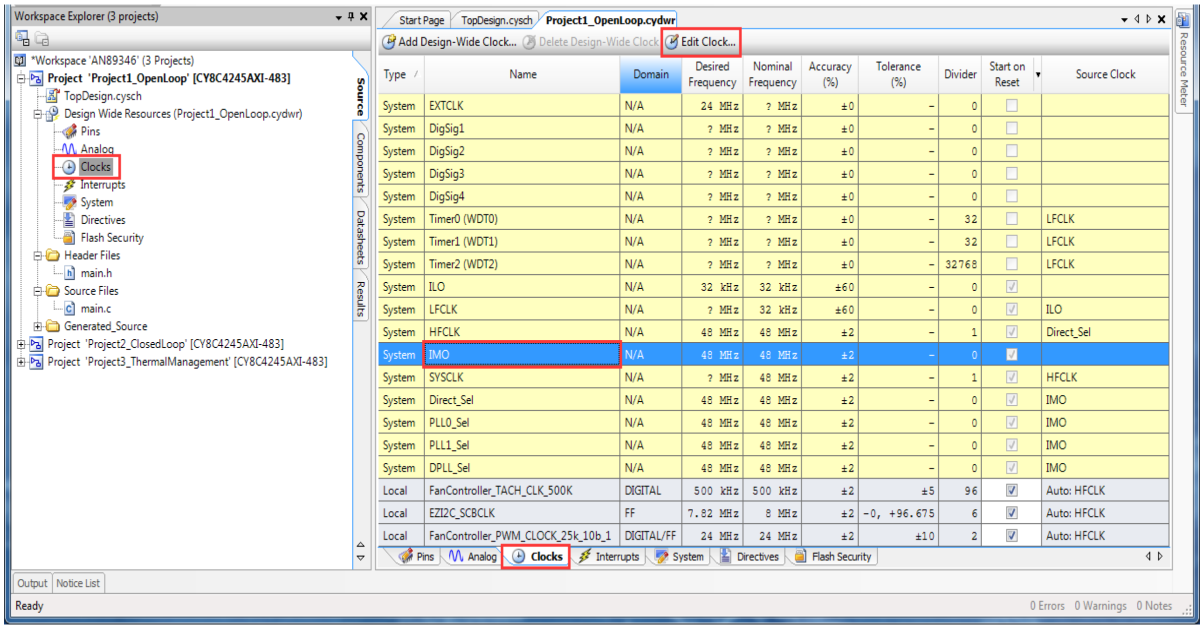
I^{2}C communication is included to allow an external I^{2}C master to send commands to update the PWM duty cycle and read data from the PSoC. An EzI2C Component, which acts as an I^{2}C slave, is placed in the design for this purpose.

Two digital input pin components with internal resistive pull-up are used to read the switch status.

6.3 Clock Settings

When operating the fan PWMs with a 10-bit resolution, the IMO must be set to 48 MHz. To configure the IMO, open the *Project1_OpenLoop.cydwr* file and go to the **Clocks** tab, as Figure 17 shows.

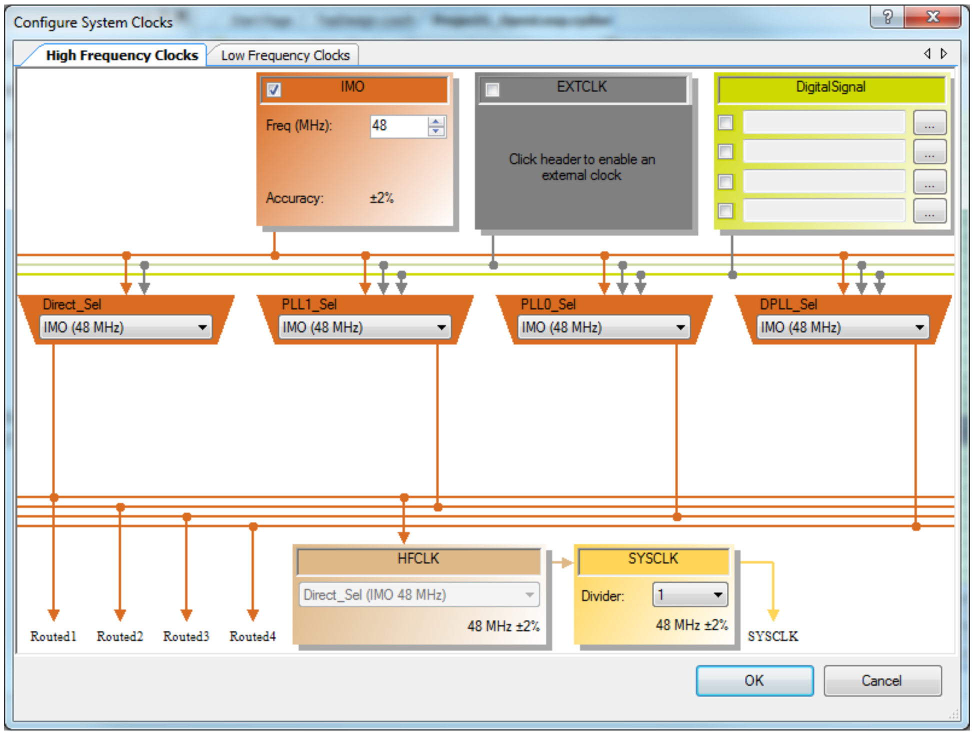
Figure 17. Clocks Tab



Type	Name	Domain	Desired Frequency	Nominal Frequency	Accuracy (%)	Tolerance (%)	Divider	Start on Reset	Source Clock
System	EXTCLK	N/A	24 MHz	? MHz	±0	-	0	☐	
System	DigSig1	N/A	? MHz	? MHz	±0	-	0	☐	
System	DigSig2	N/A	? MHz	? MHz	±0	-	0	☐	
System	DigSig3	N/A	? MHz	? MHz	±0	-	0	☐	
System	DigSig4	N/A	? MHz	? MHz	±0	-	0	☐	
System	Timer0 (WDT0)	N/A	? MHz	? MHz	±0	-	32	☐	LFCLK
System	Timer1 (WDT1)	N/A	? MHz	? MHz	±0	-	32	☐	LFCLK
System	Timer2 (WDT2)	N/A	? MHz	? MHz	±0	-	32768	☐	LFCLK
System	ILO	N/A	32 kHz	32 kHz	±60	-	0	✓	
System	LFCLK	N/A	? MHz	32 kHz	±60	-	0	✓	ILO
System	HFCLK	N/A	48 MHz	48 MHz	±2	-	1	✓	Direct_Sel
System	IMO	N/A	48 MHz	48 MHz	±2	-	0	✓	
System	SYSCLK	N/A	? MHz	48 MHz	±2	-	1	✓	HFCLK
System	Direct_Sel	N/A	48 MHz	48 MHz	±2	-	0	✓	IMO
System	PLL0_Sel	N/A	48 MHz	48 MHz	±2	-	0	✓	IMO
System	PLL1_Sel	N/A	48 MHz	48 MHz	±2	-	0	✓	IMO
System	DPLL_Sel	N/A	48 MHz	48 MHz	±2	-	0	✓	IMO
Local	FanController_TACH_CLK_500K	DIGITAL	500 kHz	500 kHz	±2	±5	96	✓	Auto: HFCLK
Local	EZI2C_SCBCLK	FF	7.82 MHz	8 MHz	±2	-0, +96.675	6	✓	Auto: HFCLK
Local	FanController_PWM_CLOCK_25k_10b_1	DIGITAL/FF	24 MHz	24 MHz	±2	±10	2	✓	Auto: HFCLK

Select **IMO** and click **Edit Clock**. This opens the **Configure System Clocks** dialog, as Figure 18 shows. Set the **IMO** clock to 48 MHz. **SYSCLK** can be set based on your requirements. For this project, set the **Divider** to 1, which sets the **SYSCLK** to 48 MHz.

Figure 18. System Clock Configuration

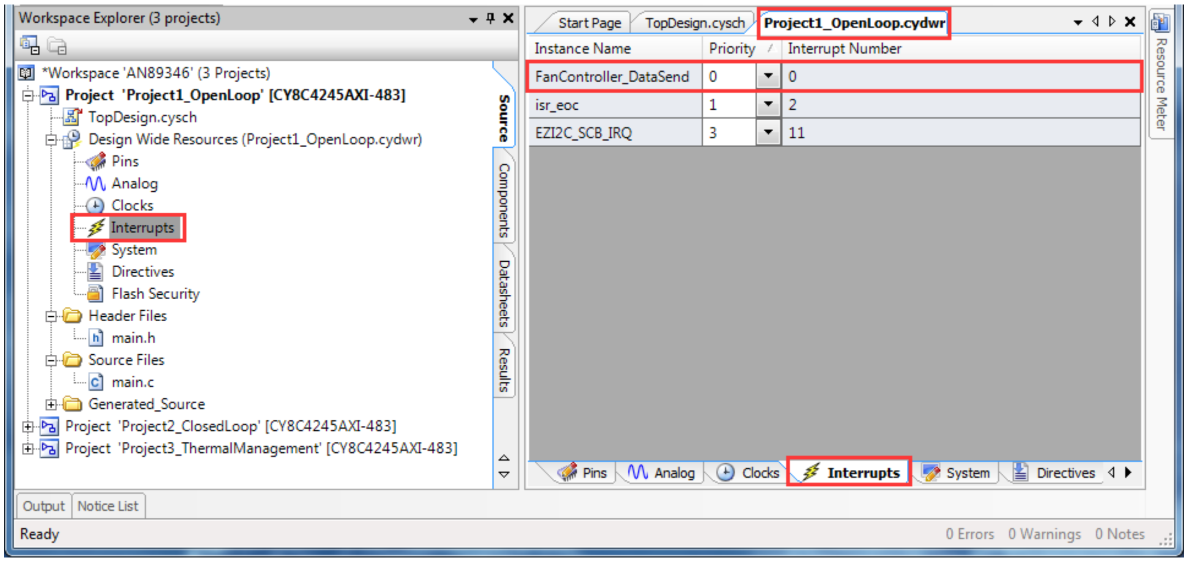


6.4 Interrupt Priorities

The FanController Component has two interrupt service routines (ISRs):

- FanController_PID_ISR - present only when the **Automatic (Firmware)** mode of the FanController Component is enabled. It executes the PID control algorithm.
- FanController_DataSend - used for transferring the measured RPM speed from the tachometer to the SRAM. The priority of this ISR should be highest in the project. To set the priority, open the *Project1_OpenLoop.cydwr* file. Go to the Interrupts tab and set the priority of FanController_DataSend interrupt to 0, which is the highest priority, as Figure 19 shows. Keep the EZI2C interrupt to lowest priority.

Figure 19. Interrupt Priority Setting



Instance Name	Priority	Interrupt Number
FanController_DataSend	0	0
isr_eoc	1	2
EZI2C_SCB_IRQ	3	11

The FanController_DataSend interrupt should read the speed result within a small time window. Failure to do so leads to an erroneous speed result. This time window is given by:

$$N_{cpu\_cycles} = F_{cpu\_clk} / F_{tach\_clk}$$

F_{cpu_clk} is the CPU clock frequency, F_{tach_clk} is the tachometer clock frequency, and N_{cpu_cycles} is the maximum latency in terms of the CPU clock cycles to move the speed result to SRAM. The tachometer clock frequency is 500 KHz. For a CPU clock frequency of 48 MHz, the maximum latency is 96 CPU clock cycles. This requires the FanController_DataSend interrupt to have the highest priority. It is also important not to disable the FanController_DataSend interrupt; make sure you are not using the CyEnterCriticalSection() API as it disables all the interrupts. Instead, disable individual interrupts when required.

6.5 Firmware Details

In the Workspace Explorer window, double-click *main.c* to examine the firmware used in this example project. The purpose of this first example is twofold:

- Capture the duty cycle to RPM relationship if the fan manufacturer's datasheet is not available.
- Get a sense of the accuracy of the fans. It is common for two "identical" fans to spin at very different speeds even when they are both driven with the PWM of same duty cycle. In later examples when we explore the closed-loop hardware control mode, we will revisit this topic.

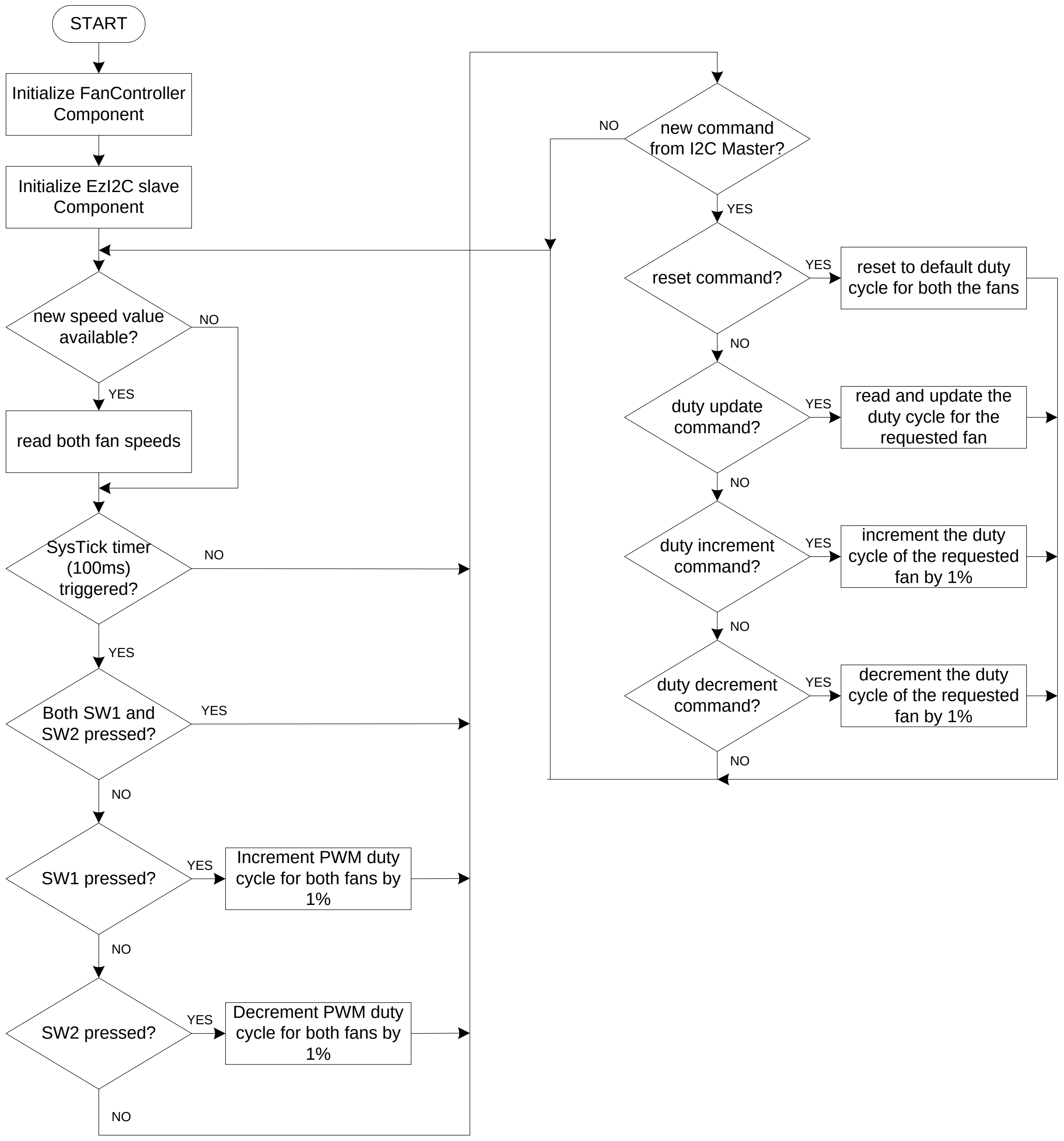
There are four commands implemented in the project – Reset, Duty Cycle Update, Duty Cycle Increment, and Duty Cycle Decrement. Each command is assigned a code, as shown in Table 2, which the external I^{2}C master can use.

Table 2. Commands for Project1_OpenLoop

Command	Code	Comments
Reset	0x0001	This command resets the duty cycle of fans to default value (25%). Default duty cycle can be changed in firmware by altering the macro DEFAULT_DUTY_CYCLE in *main.h*.
Duty Cycle Update	0x0002	Using this command, an external host can alter the duty cycle of any fan. Duty cycle count should be 100 times the required duty cycle in percent.
Duty Cycle Increment	0x0003	Using this command, an external host can increment the duty cycle of any fan. In the present code, the step size is set to 1%. This can be changed by altering the macro DUTY_STEP in *main.h*
Duty Cycle Decrement	0x0004	This command decrements the duty cycle of a particular fan by DUTY_STEP.

Figure 20 shows the firmware flowchart.

Figure 20. Firmware Flowchart


START
Initialize FanController Component
Initialize EzI2C slave Component
new speed value available?
NO
YES
read both fan speeds
SysTick timer (100ms) triggered?
NO
YES
Both SW1 and SW2 pressed?
YES
NO
SW1 pressed?
YES
Increment PWM duty cycle for both fans by 1%
NO
SW2 pressed?
YES
Decrement PWM duty cycle for both fans by 1%
NO
new command from I2C Master?
NO
YES
reset command?
YES
reset to default duty cycle for both the fans
NO
duty update command?
YES
read and update the duty cycle for the requested fan
NO
duty increment command?
YES
increment the duty cycle of the requested fan by 1%
NO
duty decrement command?
YES
decrement the duty cycle of the requested fan by 1%
NO

The EzI2C Slave Component looks like a memory device as seen by an I^{2}C master. An array is defined in the project for the EzI2C Slave Component to store different fan parameters, as shown in Figure 21.

Figure 21. Buffer for EzI2C Slave Component

SUB ADDR		
0x00	Command Code	R/W
0x02	Fan Identifier	R/W
0x04	Duty Cycle Input	R/W
0x06	Fan1 Duty Cycle	Read Only
0x08	Fan2 Duty Cycle	Read Only
0x0A	Fan1 Actual Speed	Read Only
0x0C	Fan2 Actual Speed	Read Only

All of the parameters are two bytes wide in the little endian format. The Duty Cycle Update, Increment, and Decrement commands need to send one more argument to identify the fan for that command. The FAN1 identifier is 0x01 and the FAN2 identifier is 0x02. The duty cycle has a scaling factor of 100. For example, if the required duty cycle is 1%, then send 100. The speed read from the array is in units of RPM.

Commands from the host must be in the format shown in Figure 22.

Figure 22. Command Format

	Slave address with write bit	Sub address (8 bit)	Command (16 bit)
Reset Command	<addr>	0x00	0x0100

	Slave address with write bit	Sub address (8 bit)	Command (16 bit)	Fan Identifier (16 bit)	Duty Cycle Input (16 bit)
Duty Cycle Update Command	<addr>	0x00	0x0200	FAN1 – 0x0100 FAN2 - 0x0200	<duty cycle>

	Slave address with write bit	Sub address (8 bit)	Command (16 bit)	Fan Identifier (16 bit)
Duty Cycle Increment Command	<addr>	0x00	0x0300	FAN1 – 0x0100 FAN2 - 0x0200

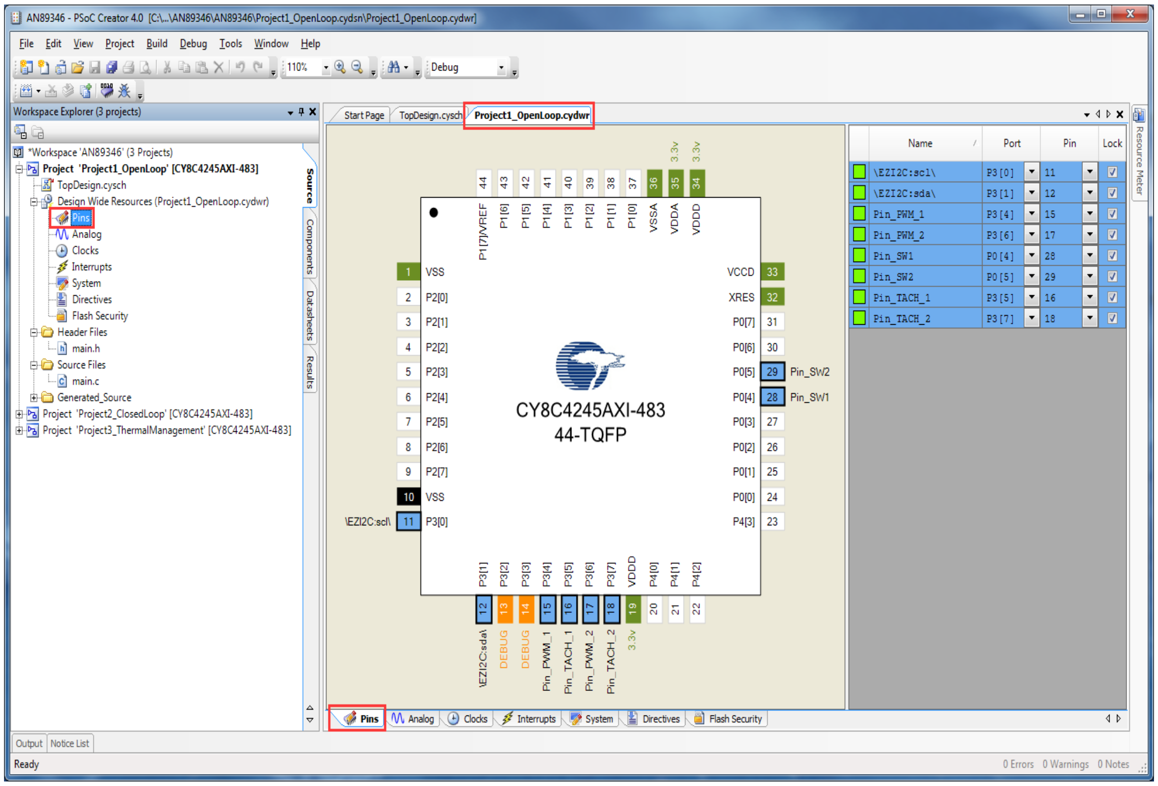
	Slave address with write bit	Sub address (8 bit)	Command (16 bit)	Fan Identifier (16 bit)
Duty Cycle Decrement Command	<addr>	0x00	0x0400	FAN1 – 0x0100 FAN2 - 0x0200

You can also use the switches to change the PWM duty cycle for both the fans. Switch SW1 increases the duty cycle by 1% and switch SW2 decreases it by 1%.

6.6 Test Setup

The following section describes the test setup using the kits described in Cypress Development Kits for Thermal Management. If you are not using these kits, you can still test the project. Ensure that you make proper pin connections in your test board. For the pins selected in this project, refer to the Project1_*OpenLoop.cydwr* file, as Figure 23 shows.

Figure 23. Project1_OpenLoop - Pins

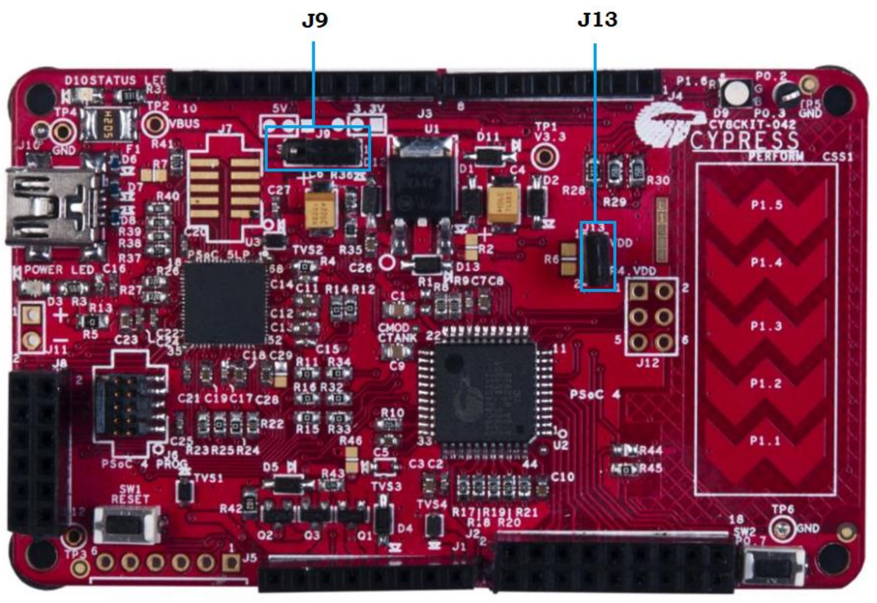


AN89346 - PSoC Creator 4.0 [C:\...\AN89346\AN89346\Project1_OpenLoop.cydsn\Project1_OpenLoop.cydwr]

Name	Port	Pin	Lock
\EZI2C:scl\	P3[0]	11	✓
\EZI2C:sda\	P3[1]	12	✓
Pin_PWM_1	P3[4]	15	✓
Pin_PWM_2	P3[6]	17	✓
Pin_SW1	P0[4]	28	✓
Pin_SW2	P0[5]	29	✓
Pin_TACH_1	P3[5]	16	✓
Pin_TACH_2	P3[7]	18	✓

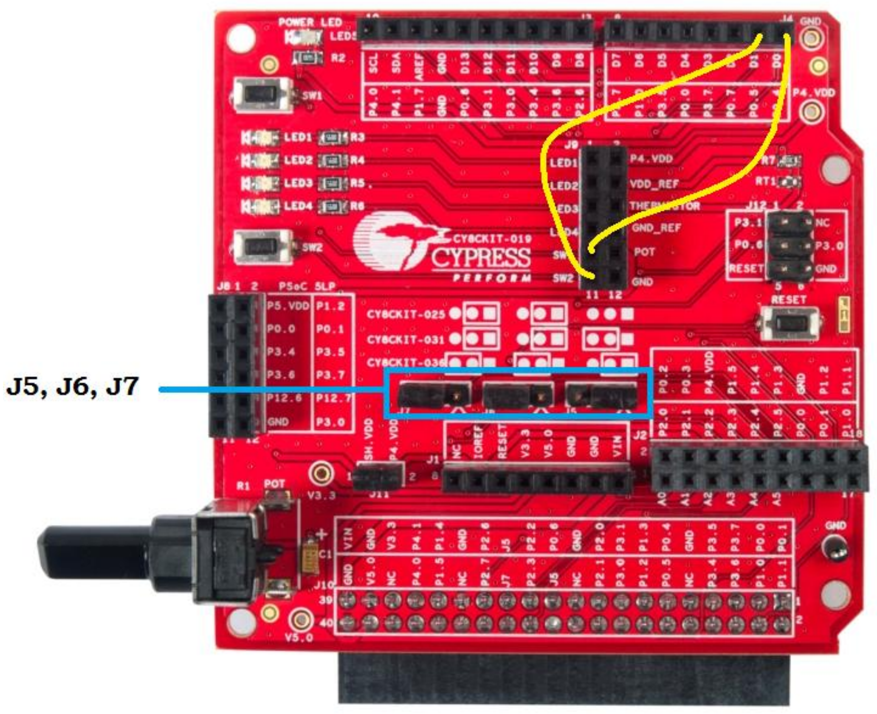
1. Connect jumper J9 of the CY8CKIT-042 Pioneer kit to the 3.3-V position and ensure that J13 is connected. See Figure 24 for jumper positions.

Figure 24. CY8CKIT-042 Jumper Settings



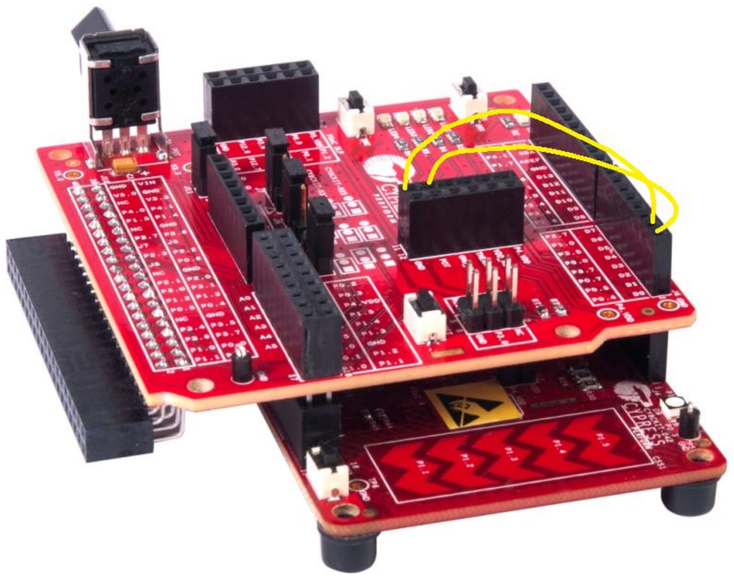
2. Ensure jumpers J5, J6, and J7 on the CY8CKIT-019 PSoC Shield Adapter kit are configured to work with the CY8CKIT-036 Kit. See Figure 25 for jumper positions.
3. On the CY8CKIT-019, connect a wire between SW1 in header J9 and P0[4] of J4. Also, connect a wire between SW2 in header J9 and P0[5] of J4.This is to make connections between the switches and the PSoC pins.

Figure 25. CY8CKIT-019 Setup



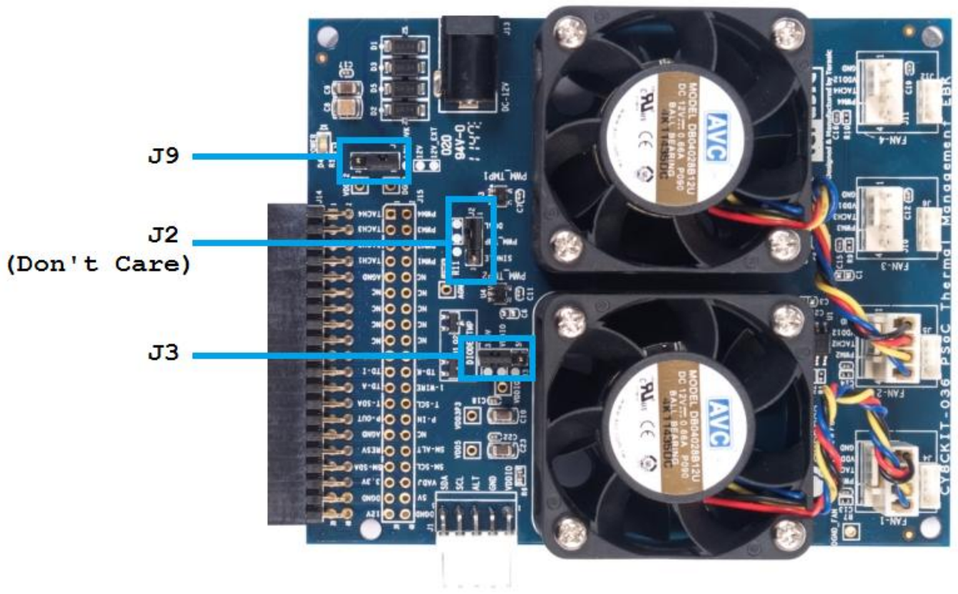
4. Plug the CY8CKIT-019 on to the CY8CKIT-042, as shown in Figure 26.

Figure 26. CY8CKIT-042 and CY8CKIT-019



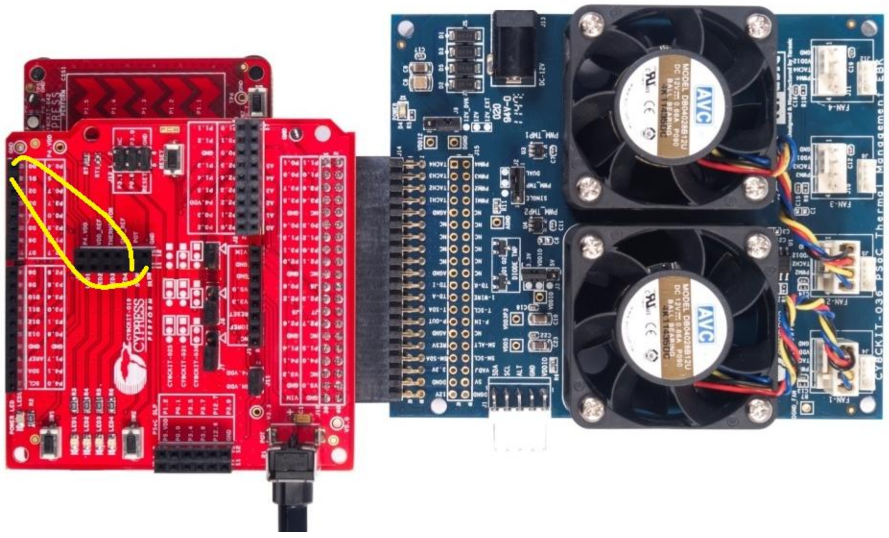
5. Set the power jumper J9 on the CY8CKIT-036 EBK board to 12V_EXT and the VDDIO selection jumper J3 to 3.3 V, as shown in Figure 27.

Figure 27. CY8CKIT-036 Setup



6. Connect J14 of the CY8CKIT-036 to J10 of CY8CKIT-019, as shown in Figure 28.

Figure 28. Complete Setup



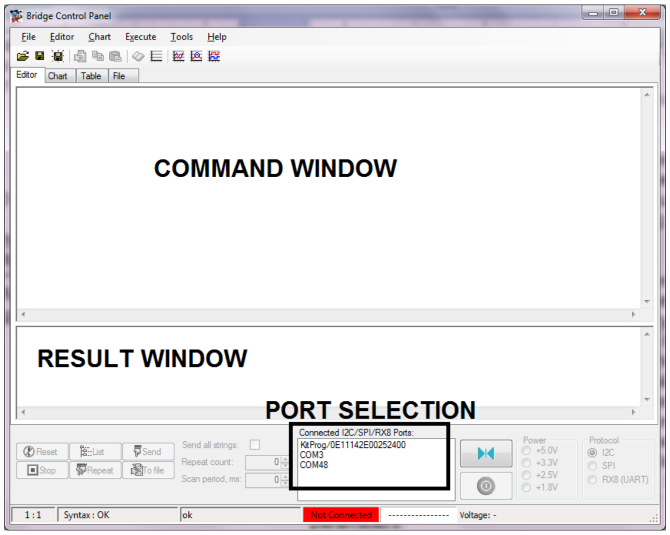
6.7 Test Procedure

If you are using the Cypress kits, follow these steps:

1. Make the hardware setup as shown in the section Test Setup.
2. Connect the USB cable and program the PSoC 4 device, by selecting **Debug** > **Program** in the PSoC Creator menu.
3. Power the CY8CKIT-036 Kit with the 12 V/2A adapter. Notice that both the fans start rotating at some speed. The default PWM duty cycle is set to 25%.
4. Press the switch SW1 and notice that the speed of both fans increases. On pressing switch SW2, the speed of both fans decreases.
5. To find the actual speed of the fans, use the Cypress Bridge Control Panel (BCP) software. The BCP is a graphical user interface used to communicate with an I^{2}C slave through an I^{2}C-USB Bridge. You can use the CY8CKIT-042 directly because the onboard PSoC 5LP acts as an I^{2}C-USB Bridge. If you are new to BCP, see its Help documentation.

   This software is installed automatically when the PSoC Programmer software is installed. You can open the software by going to **Start** > **All Programs** > **Cypress** > **Bridge Control Panel**. The GUI consists of three main controls – the port selection window, the command window, and the result window, as Figure 29 shows.

Figure 29. Bridge Control Panel

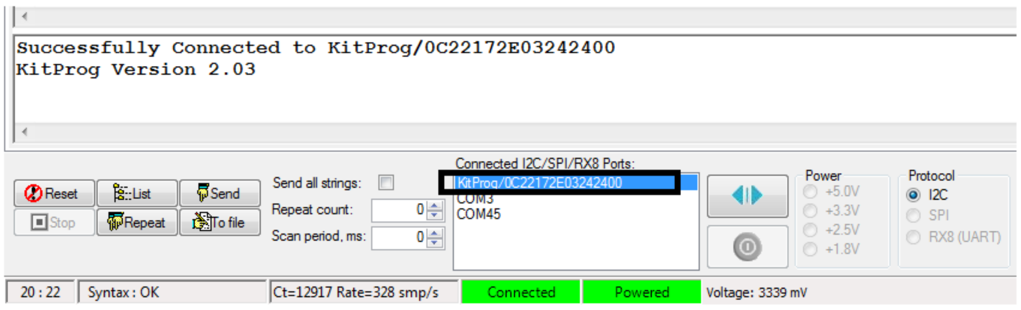


Connected I2C/SPI/RX8 Ports:
KitProg/0E11142E00252400
COM3
COM48

1 : 1 | Syntax : OK | ok | Not Connected | --------------- | Voltage: -

Notice that the bridge (the PSoC 5LP on the CY8CKIT-042 board) is listed in the **Connected Ports** window when you connect the PSoC 4 Pioneer Kit to your PC, as shown in Figure 30.

Figure 30. Connected Bridge Devices



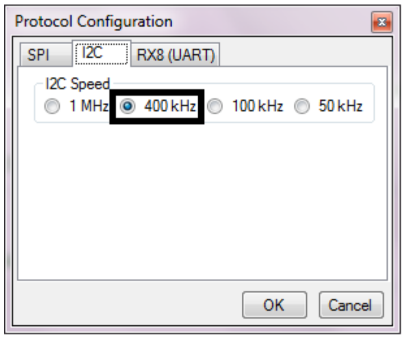
```
Successfully Connected to KitProg/0C22172E03242400
KitProg Version 2.03
```

Send all strings: ☐
Repeat count: 0
Scan period, ms: 0
Connected I2C/SPI/RX8 Ports: KitProg/0C22172E03242400, COM3, COM45
Protocol: I2C

20 : 22 | Syntax : OK | Ct=12917 Rate=328 smp/s | Connected | Powered | Voltage: 3339 mV

6. Click **KitProg/<ID>** to select the device
7. Now configure the protocol to 400 kHz, using the menu option **Tools** > **Protocol Configuration** > **I2C**, as shown in Figure 31.

Figure 31. I^{2}C Protocol Configuration

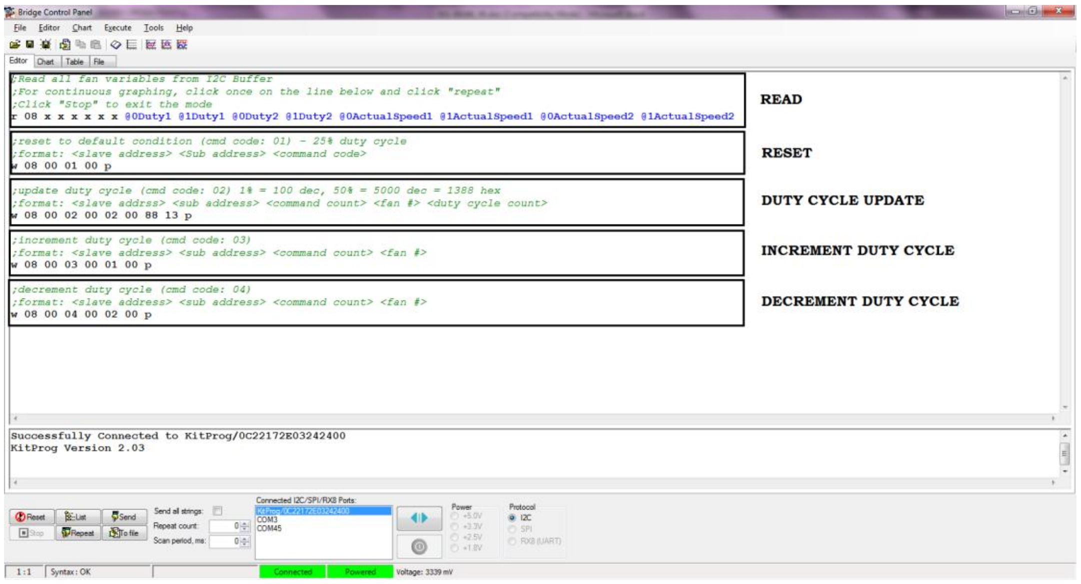


Protocol Configuration
I2C Speed: ( ) 1 MHz [✓ 400 kHz] ( ) 100 kHz ( ) 50 kHz

8. To send a command, you can either start typing in the command window or use the *.iic* file, which contains preset commands. The *.iic* file *OpenLoopControl.iic* is provided for this project. Use the **File** > **Open File** menu option to open it. This file is available in the following project zip file path: `/AN89346/BridgeFiles/OpenLoopControl.iic`

The command window is filled with instructions, as shown in *Figure 32.*

Figure 32. I2C Commands


Bridge Control Panel
File Editor Chart Execute Tools Help
Editor Chart Table File
Reset List Send Stop Repeat To file
Send all strings:
Repeat count: 0
Scan period, ms: 0
Connected I2C/SPI/RX8 Ports:
Power +5.0V +3.3V +2.5V +1.8V
Protocol I2C SPI RX8 (UART)
1:1 Syntax : OK Connected Powered Voltage: 3339 mV

```
;Read all fan variables from I2C Buffer
;For continuous graphing, click once on the line below and click "repeat"
;Click "Stop" to exit the mode
r 08 x x x x x x @0Duty1 @1Duty1 @0Duty2 @1Duty2 @0ActualSpeed1 @1ActualSpeed1 @0ActualSpeed2 @1ActualSpeed2
```
READ

```
;reset to default condition (cmd code: 01) - 25% duty cycle
;format: <slave address> <Sub address> <command code>
w 08 00 01 00 p
```
RESET

```
;update duty cycle (cmd code: 02) 1% = 100 dec, 50% = 5000 dec = 1388 hex
;format: <slave addrss> <sub address> <command count> <fan #> <duty cycle count>
w 08 00 02 00 02 00 88 13 p
```
DUTY CYCLE UPDATE

```
;increment duty cycle (cmd code: 03)
;format: <slave address> <sub address> <command count> <fan #>
w 08 00 03 00 01 00 p
```
INCREMENT DUTY CYCLE

```
;decrement duty cycle (cmd code: 04)
;format: <slave address> <sub address> <command count> <fan #>
w 08 00 04 00 02 00 p
```
DECREMENT DUTY CYCLE

```
Successfully Connected to KitProg/0C22172E03242400
KitProg Version 2.03
```

Connected I2C/SPI/RX8 Ports: KitProg/0C22172E03242400, COM3, COM45

The **ins**tructions include I2C read and write commands; their formats are:

r <slave address> <var*iables for storing the data>*

***w <slave add*ress> <sub address> <data to be sent>**

There is a single I2C read command to get the current duty cycle and actual speed of both the fans. There are four write instructions for the commands listed in Table 2 on page 13.

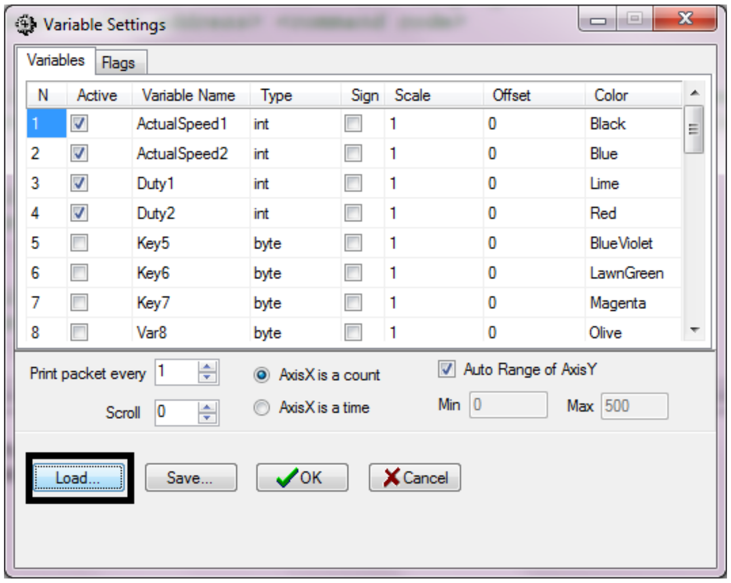
9. For the read command, you have the option to store the read data in **variables**. You can define the variables in the **Chart** > **Variable Settings** menu option or directly load *.ini* files when available. The *.ini* files are used to save information on variable names and types. An *.ini* file *OpenLoopControl.ini* is also provided for this project. Use the **Load** button in the **Variable Settings** window and browse to the *.ini* file. You can find the project *.ini* file in:

```
/AN89346/BridgeFiles/OpenLoopFanControl.ini
```

After selecting the file, click **OK**.

Figure 33 shows the variables used for the Project1_OpenLoop.

Figure 33. I2C Variables

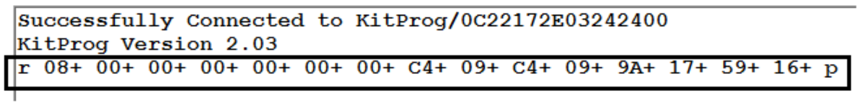


N	Active	Variable Name	Type	Sign	Scale	Offset	Color
1	✓	ActualSpeed1	int	☐	1	0	Black
2	✓	ActualSpeed2	int	☐	1	0	Blue
3	✓	Duty1	int	☐	1	0	Lime
4	✓	Duty2	int	☐	1	0	Red
5	☐	Key5	byte	☐	1	0	BlueViolet
6	☐	Key6	byte	☐	1	0	LawnGreen
7	☐	Key7	byte	☐	1	0	Magenta
8	☐	Var8	byte	☐	1	0	Olive

Print packet every 1
Scroll 0
[✓ AxisX is a count] AxisX is a time
✓ Auto Range of AxisY
Min 0 Max 500

10. To execute the read command, place the cursor on the read command (line 4) and press **Enter**. The read data is displayed in the result window, as shown in Figure 34.

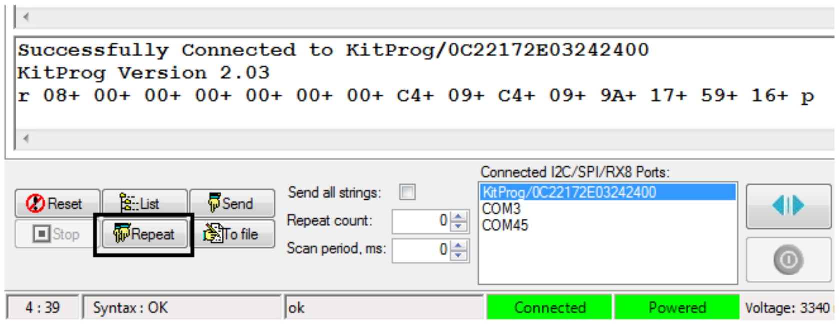
Figure 34. Read Bytes



```
Successfully Connected to KitProg/0C22172E03242400
KitProg Version 2.03
r 08+ 00+ 00+ 00+ 00+ 00+ 00+ C4+ 09+ C4+ 09+ 9A+ 17+ 59+ 16+ p
```

The data is assigned to the declared variables. Now execute this command continuously by clicking the **Repeat** button, as shown in Figure 35. Go to the **Chart** tab to see the variables plotted over time.

Figure 35. Repeat Button



```
Successfully Connected to KitProg/0C22172E03242400
KitProg Version 2.03
r 08+ 00+ 00+ 00+ 00+ 00+ 00+ C4+ 09+ C4+ 09+ 9A+ 17+ 59+ 16+ p
```

Send all strings:
Repeat count: 0
Scan period, ms: 0
Connected I2C/SPI/RX8 Ports:
KitProg/0C22172E03242400
COM3
COM45
4 : 39 | Syntax : OK | ok | Connected | Powered | Voltage: 3340

11. Press the switch SW1 and observe the fan speed in the chart. Repeat the experiment with SW2. Notice the difference between the two fan speeds when the same PWM duty cycle is applied; see Figure 36.

Figure 36. Read Command Results - Graph

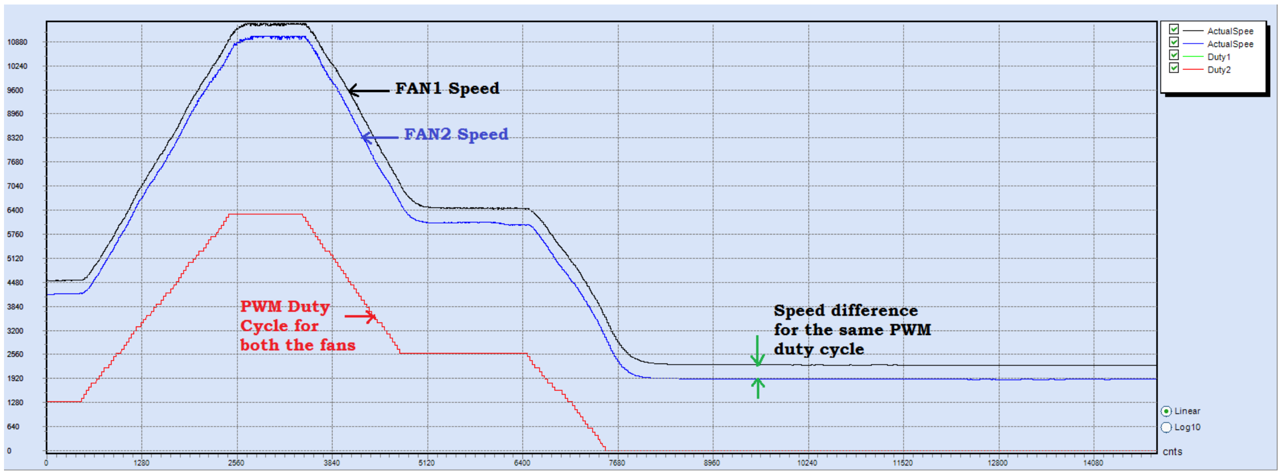


Figure 36 also shows that with zero PWM duty cycle, the fan speed is non zero.

12. To execute other commands, place the cursor on the row containing the command and press **Enter**. See the command format shown in Figure 22.

13. If your fan's datasheet does not mention the relation between the PWM duty cycle and the fan speed, vary the duty cycle using the increment/decrement commands or the switches, and read the speed by executing the read command. You can then reconfigure the FanController Component with this data (Figure 16).

    Note that when the PSoC 4 device is reset or it is power cycled or it is being programmed, the PWM pins are tri-stated. This causes the fan to run at the maximum speed if the CY8CKIT-036 Thermal Management EBK is powered. If you want the fan to turn off during these conditions, connect external pull-down resistors of around 4.7 KΩ to the PWM pins. The PWM pins are accessible on header J3 of the CY8CKIT-019 PSoC Shield Adapter.

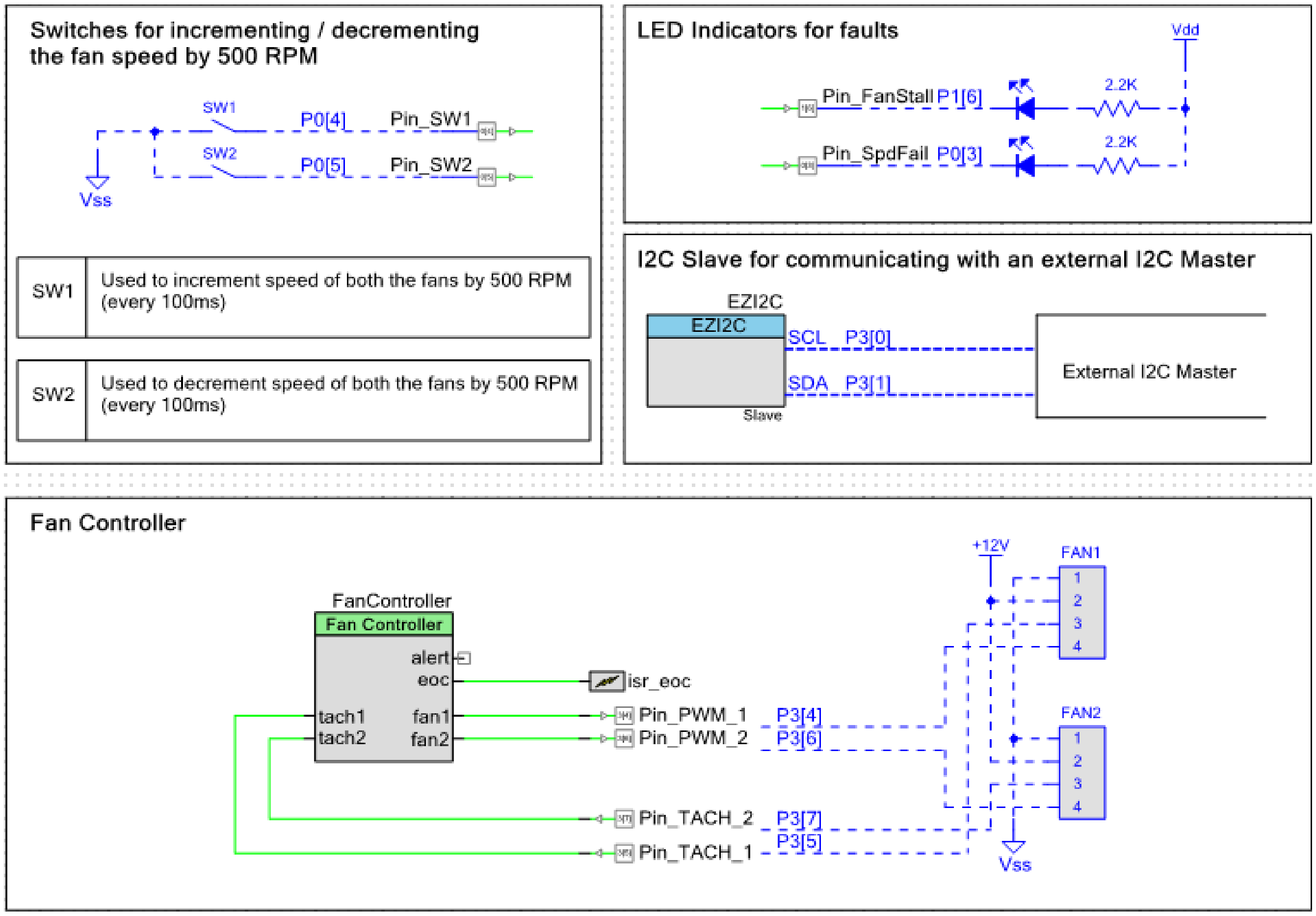
7 Project2_ClosedLoop

Close the files *main.c* and *TopDesign.cysch* for the first project. Go back to the Workspace Explorer window, right-click Project 'Project2_ClosedLoop', and select **Set As Active Project**.

This project uses the FanController Component to operate the fans with closed-loop speed control. In addition, the alert feature, which includes fan stall error and speed regulation error, is demonstrated in this project.

Open the TopDesign for this project. Figure 37 shows the *TopDesign* schematic.

Figure 37. Project2_ClosedLoop - TopDesign



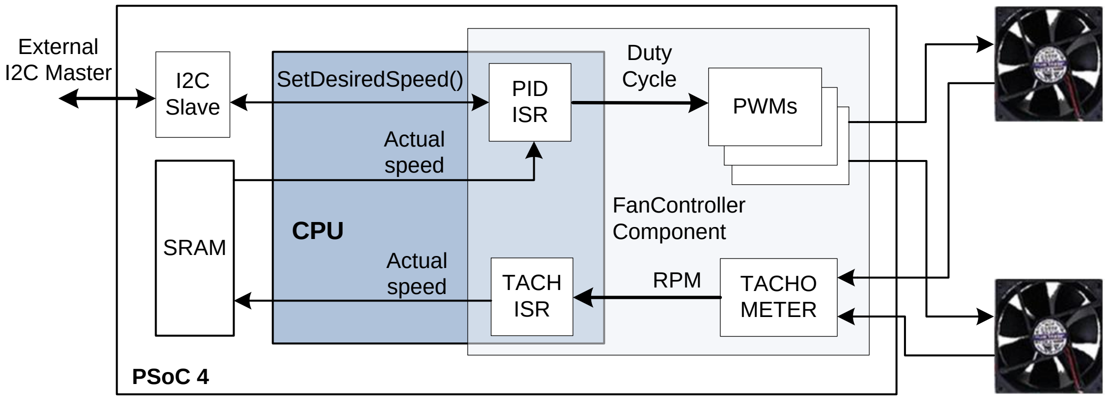
SW1	Used to increment speed of both the fans by 500 RPM (every 100ms)
SW2	Used to decrement speed of both the fans by 500 RPM (every 100ms)

The closed-loop control mode of the FanController Component measures the current fan speed and updates the duty cycle to achieve the required speed. The Component uses the **P**roportional, **I**ntegral, **D**erivative (PID) loop, formed using the hardware blocks and some firmware in the interrupt service routines (ISRs), to get the desired fan response. PID parameters affect the way the fan responds to the speed change request. The output response can be analyzed for rise time, peak overshoot, steady state error, and stability. Practically, it is not possible to get the best values of all these characteristics in a design – there is a tradeoff between these characteristics. A proper tuning of PID constants can provide the best combination suited for an application. PID Tuning on page 30 section discusses on how to select these constants.

As shown in Figure 38, two ISRs are present in the FanController Component. One reads the current speed (TACH ISR). The other runs the PID control algorithm (PID ISR), which is executed periodically to generate the new PWM duty cycle. The fan speed is set using the API function SetDesiredSpeed(). There are two differences in this project as compared to Project1_OpenLoop.

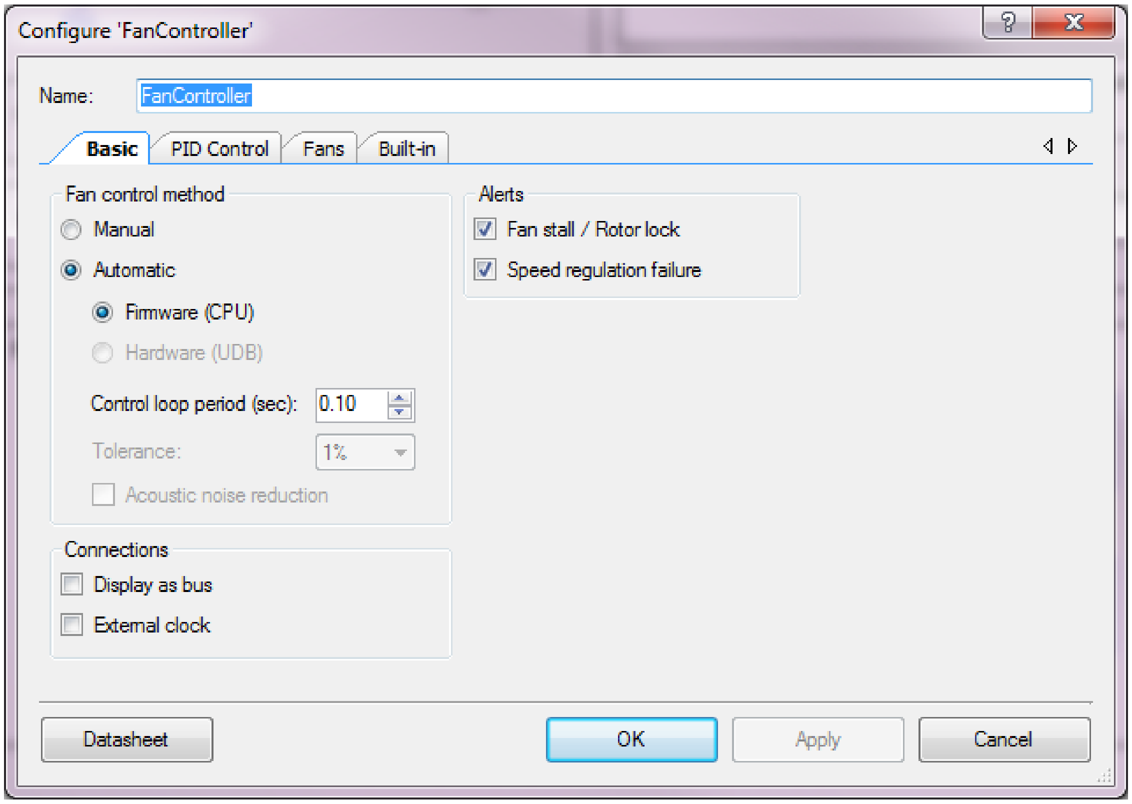
- The FanController Component is configured to do PID control for the fan speed
- Pin components are placed to drive LEDs for fan alerts

Figure 38. Automatic FanController Mode - Block Diagram



Let us start by configuring the FanController Component; see Figure 39.

Figure 39. Automatic Mode in FanController Component


Configure 'FanController'

Name: FanController

Basic | PID Control | Fans | Built-in

Fan control method
- Manual
- [✓ Automatic]
  - [✓ Firmware (CPU)]
  - Hardware (UDB)

Control loop period (sec): 0.10

Tolerance: 1%

Acoustic noise reduction

Alerts
- [✓] Fan stall / Rotor lock
- [✓] Speed regulation failure

Connections
- [ ] Display as bus
- [ ] External clock

Open the Component Configuration Tool and go to the **Basic** tab. Select **Automatic.** This enables closed-loop fan speed control. You can also set the **control loop period**. This sets how frequently the PID control loop executes. It can be set in increments of 10 ms. It is important that the control loop period be greater than the time required to calculate the speed of all of the fans.

The worst-case speed measurement time for a single fan depends on its lowest RPM as Equation 1 shows.

Equation 1. Minimum Control Loop Period

$$T_{FAN} = 1.75 \times (\frac{60}{RPM_{min}})$$

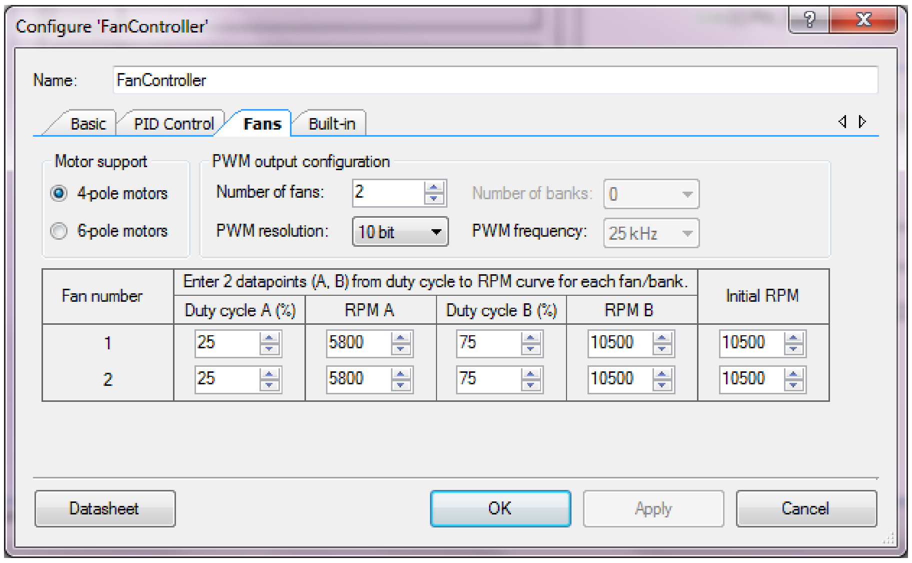
In this project, assuming minimum fan speed to be 2500 RPM, T_{FAN} is 42 ms. Therefore, for two fans the minimum **control loop period** is 84 ms. Set the **control loop period** to 100 ms for this project.

The alert feature of FanController is used in this project – it enables the detection of stalled fan rotor and speed regulation failure. Speed regulation failure can happen due to a fault in the fan or insufficient voltage supply. Stalling can happen for mechanical reasons.

A particular fault can be detected by polling the status using the API functions, GetFanSpeedStatus() and GetFanStallStatus(). In the project, GPIOs are assigned to drive LEDs to indicate the two faults. Pin P0[3] is held low when a stall is detected. Pin P1[6] is held low when a speed regulation failure is detected. Note that the pins driving the LEDs are configured in this project to be in an active low configuration. If you are using the CY8CKIT-042 PSoC 4 Pioneer kit, an RGB LED in active low configuration is already present onboard. This project uses two of those LEDs. Blinking of the blue LED indicates a speed regulation failure and that of the red LED indicates a fan stall error.

Now click the **Fans** tab (shown in Figure 40 for reference). If you measured the duty cycle and speed parameters for your fans in the first example project (Step 13 in the Test Procedure of first project), you can enter that data now. The parameters can be different for each fan. This allows the FanController Component to work with any combination of fans, so you have more flexibility to mix and match different types of fans to meet system cooling needs. Note that the values entered will affect the initial fan speed at power-up or after a requested speed change. However, since the fan's actual speed is measured and used in the control loop, the fan speed will be regulated to the set value even if these parameters are not correct.

Figure 40. Project1_ClosedLoop - Fan Configuration



Name: FanController

Motor support: [✓ 4-pole motors] 6-pole motors

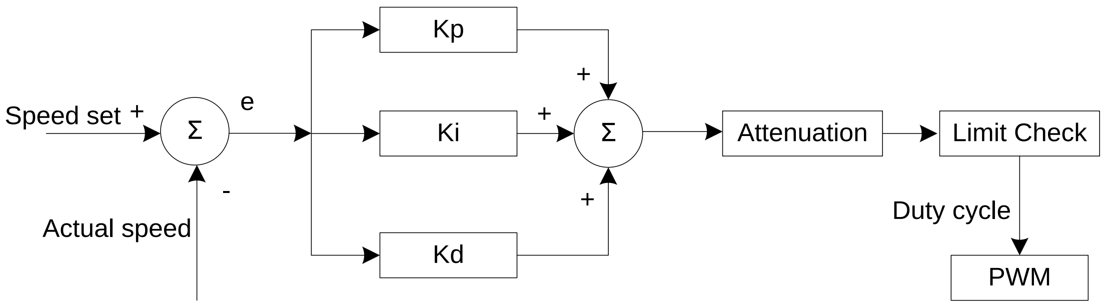
PWM output configuration
Number of fans: 2
Number of banks: 0
PWM resolution: 10 bit
PWM frequency: 25 kHz

Fan number	Enter 2 datapoints (A, B) from duty cycle to RPM curve for each fan/bank.				Initial RPM
	Duty cycle A (%)	RPM A	Duty cycle B (%)	RPM B	
1	25	5800	75	10500	10500
2	25	5800	75	10500	10500

Click the **PID Control** tab. PID, that is, proportional, integral, and derivative, constants are set here; see Figure 41. PID controllers are used to achieve output response specifications such as rise time, stability, overshoot, and steady state error.

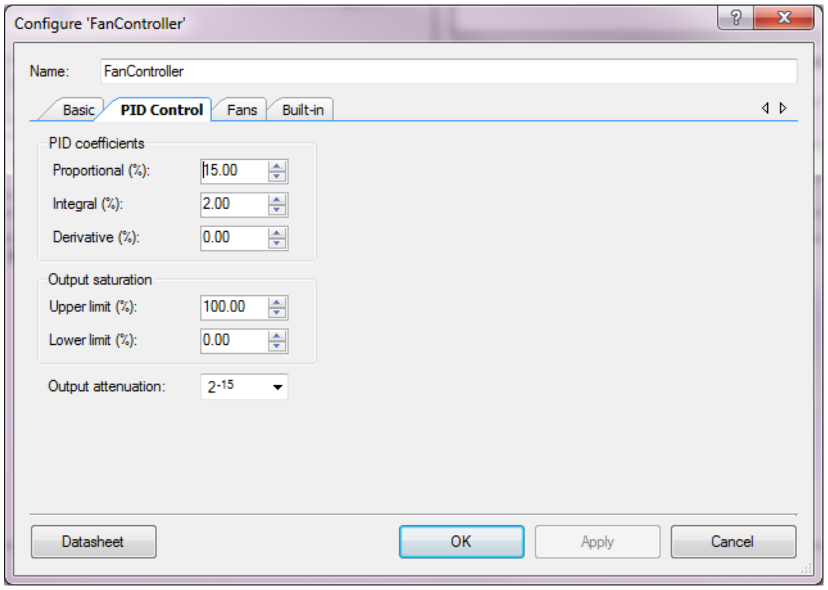
To make it simple, PID constants are set in the GUI as a percentage of a 12-bit number (4096). The Component calculates error in the fan speed and applies a PID algorithm with the coefficients set in the project. The processed PID result is then passed through an attenuation set by the **Output attenuation** parameter. Higher the **Output attenuation**, slower the output response will be. The output after attenuation is checked for the limits set in **Output saturation.** This parameter limits the duty cycle of a PWM. The resultant count is then loaded into a PWM duty cycle register.

Figure 41. PID Implementation in FanController Component



Set the parameters as given in Figure 42 for now. At the end of this section, the effects of each parameter are demonstrated.

Figure 42. PID Configuration



Name: FanController

PID coefficients

Proportional (%): 15.00

Integral (%): 2.00

Derivative (%): 0.00

Output saturation

Upper limit (%): 100.00

Lower limit (%): 0.00

Output attenuation: 2^{-15}

This completes the FanController Component configuration.

7.1 Firmware Details

In the **Workspace Explorer**, double-click *main.c* file to examine the firmware used in this example project. Similar to the Project1_OpenLoop, this project also uses I^{2}C commands and switches. However, instead of sending the duty cycle, the required speed is sent in this project.

Table 3 shows the I^{2}C commands:

Table 3. Project2_ClosedLoop- I^{2}C Commands

Command	Code	Comments
Reset	0x0001	This command resets fan speed to default value of 4500 RPM. Default speed can be changed in firmware by altering the macro INIT_RPM in *main.h*.
Speed update	0x0002	Using this command, an external host can alter the fan speed of any fan. Fan speed input should be in RPM.
Speed Increment	0x0003	Using this command, an external host can increment the speed of any fan. In the present code, step size is set to 500RPM. This can be changed by altering the macro RPM_STEP in *main.h*
Speed Decrement	0x0004	This command decrements the speed of a particular fan by RPM_STEP.

An array is defined in the project for the EzI2C Slave Component to store different fan parameters, as shown in Figure 43. All values are little endian. An external I^{2}C host should send the commands in the format, as shown in Figure 44.

Figure 43. Buffer for EzI2C Slave Component

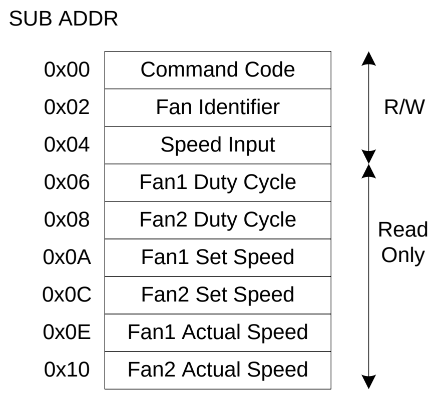


Figure 44. Project2_ClosedLoop - I^{2}C Command Format

Reset Command

Slave address with write bit	Sub address (8 bit)	Command (16 bit)
<addr>	0x00	0x0100

Speed Update Command

Slave address with write bit	Sub address (8 bit)	Command (16 bit)	Fan Identifier (16 bit)	Speed Input (16 bit)
<addr>	0x00	0x0200	FAN1 – 0x0100 FAN2 - 0x0200	<Speed RPM>

Speed Increment Command

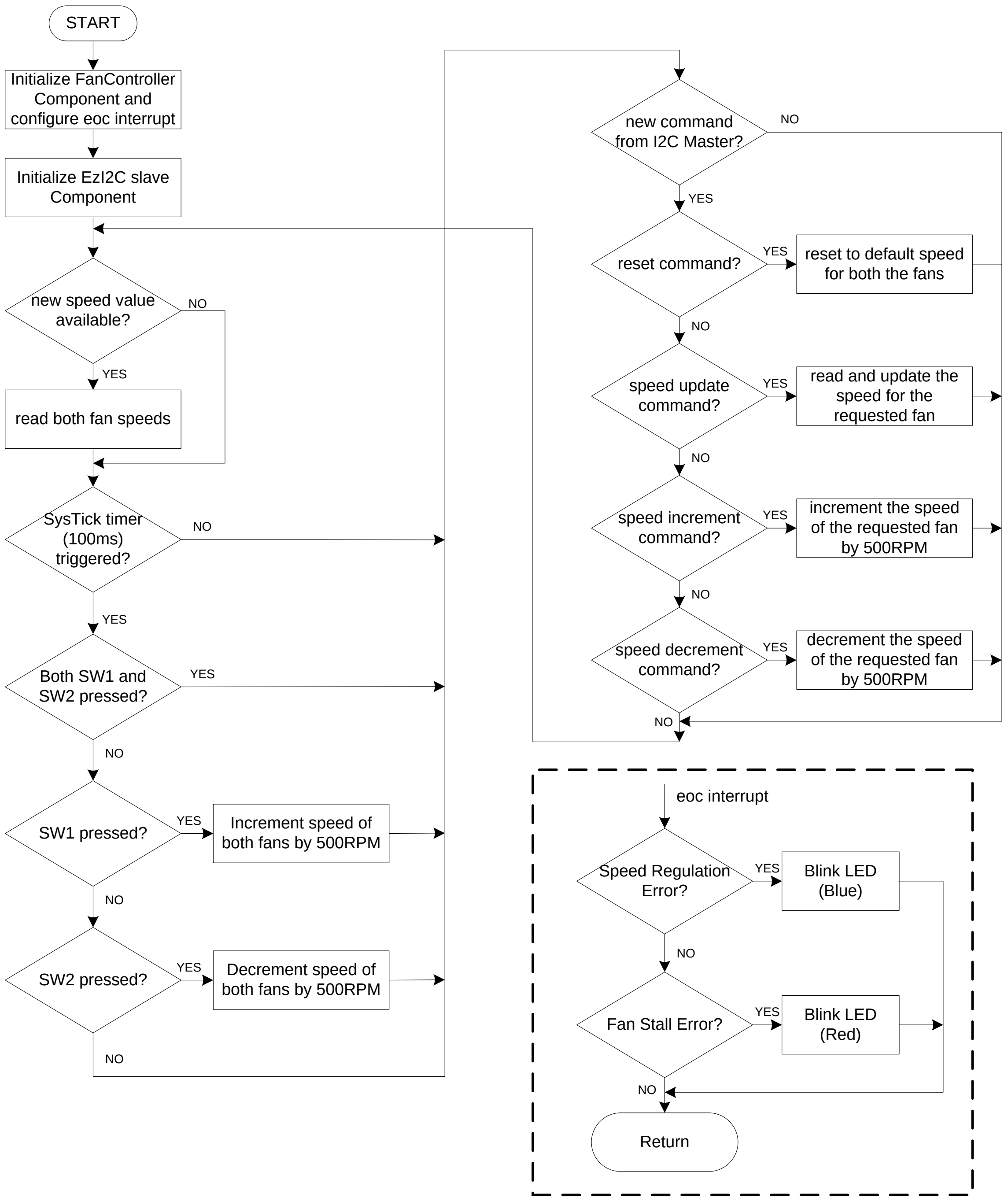
Slave address with write bit	Sub address (8 bit)	Command (16 bit)	Fan Identifier (16 bit)
<addr>	0x00	0x0300	FAN1 – 0x0100 FAN2 - 0x0200

Speed Decrement Command

Slave address with write bit	Sub address (8 bit)	Command (16 bit)	Fan Identifier (16 bit)
<addr>	0x00	0x0400	FAN1 – 0x0100 FAN2 - 0x0200

Figure 45 shows the firmware flowchart.

Figure 45. Firmware Flowchart


START
Initialize FanController Component and configure eoc interrupt
Initialize EzI2C slave Component
new speed value available?
NO
YES
read both fan speeds
SysTick timer (100ms) triggered?
NO
YES
Both SW1 and SW2 pressed?
YES
NO
SW1 pressed?
YES
Increment speed of both fans by 500RPM
NO
SW2 pressed?
YES
Decrement speed of both fans by 500RPM
NO
new command from I2C Master?
NO
YES
reset command?
YES
reset to default speed for both the fans
NO
speed update command?
YES
read and update the speed for the requested fan
NO
speed increment command?
YES
increment the speed of the requested fan by 500RPM
NO
speed decrement command?
YES
decrement the speed of the requested fan by 500RPM
NO
eoc interrupt
Speed Regulation Error?
YES
Blink LED (Blue)
NO
Fan Stall Error?
YES
Blink LED (Red)
NO
Return

7.2 Test Setup:

Do the same Test Setup that was done in the first project.

7.3 Test procedure:

If you are using the CY8CKIT-042, follow the procedure given for the first project, but use the following BCP files:

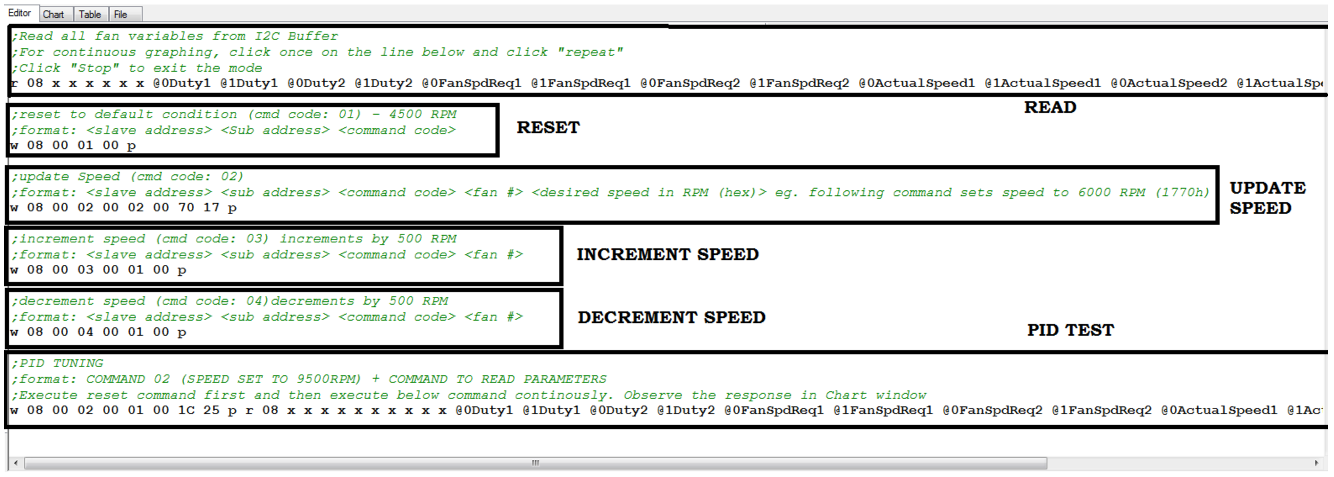
ClosedLoopControl.iic

ClosedLoopControl.ini

These files are provided in the zip file and can be found in `/AN89346/BridgeFiles/`

As shown in Figure 46, this command script has a single read command to read the set speed, the actual speed, and the current PWM duty cycle of both fans. Four write commands exist to send the reset, update speed, increment speed, and decrement speed commands. You can also use switches SW1 and SW2 of the CY8CKIT-019 for incrementing/decrementing the fan speed.

Figure 46. Project2_ClosedLoop - I^{2}C Commands



```
;Read all fan variables from I2C Buffer
;For continuous graphing, click once on the line below and click "repeat"
;Click "Stop" to exit the mode
r 08 x x x x x x @0Duty1 @1Duty1 @0Duty2 @1Duty2 @0FanSpdReq1 @1FanSpdReq1 @0FanSpdReq2 @1FanSpdReq2 @0ActualSpeed1 @1ActualSpeed1 @0ActualSpeed2 @1ActualSp

;reset to default condition (cmd code: 01) - 4500 RPM
;format: <slave address> <Sub address> <command code>
w 08 00 01 00 p

;update Speed (cmd code: 02)
;format: <slave address> <sub address> <command code> <fan #> <desired speed in RPM (hex)> eg. following command sets speed to 6000 RPM (1770h)
w 08 00 02 00 02 00 70 17 p

;increment speed (cmd code: 03) increments by 500 RPM
;format: <slave address> <sub address> <command code> <fan #>
w 08 00 03 00 01 00 p

;decrement speed (cmd code: 04)decrements by 500 RPM
;format: <slave address> <sub address> <command code> <fan #>
w 08 00 04 00 01 00 p

;PID TUNING
;format: COMMAND 02 (SPEED SET TO 9500RPM) + COMMAND TO READ PARAMETERS
;Execute reset command first and then execute below command continously. Observe the response in Chart window
w 08 00 02 00 01 00 1C 25 p r 08 x x x x x x x x x x @0Duty1 @1Duty1 @0Duty2 @1Duty2 @0FanSpdReq1 @1FanSpdReq1 @0FanSpdReq2 @1FanSpdReq2 @0ActualSpeed1 @1Ac
```

Figure 47 shows the control loop in action. Notice that it requires different PWM duty cycles to achieve the same speed in different fans. Therefore, it is important in thermal management systems to operate in a closed-loop fan control mode.

Figure 47. Graphs - Closed Loop

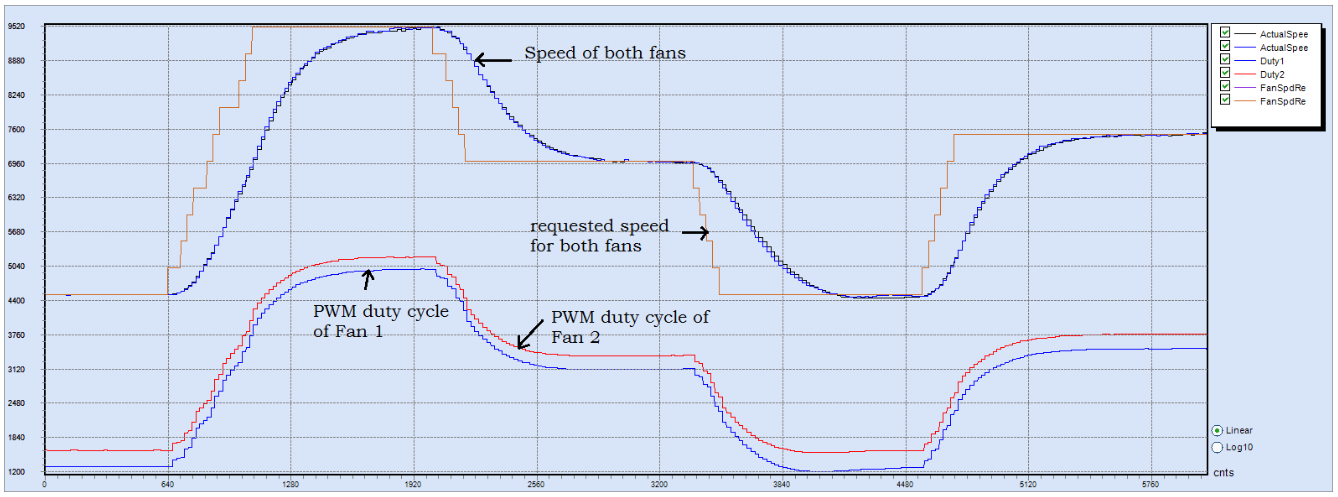


Figure 46 shows one more command, PID TEST, which is used for PID tuning. The next section discusses this command in detail.

7.3.1 PID Tuning:

Each of the PID parameters – **P**roportional, **I**ntegral, and **D**erivative – affects the output response in a particular way.

- The **proportional** parameter helps to achieve faster response but a very high value leads to high overshoot and instability.
- The **integral** parameter is similar to the proportional parameter but one major advantage is that it makes the steady state error go to zero. However, a higher integral parameter leads to overshoot.
- The **derivative** parameter helps to reduce the overshoot and settling time. It is usually minimized because it tends to amplify the noise in the error signal, leading to instability.

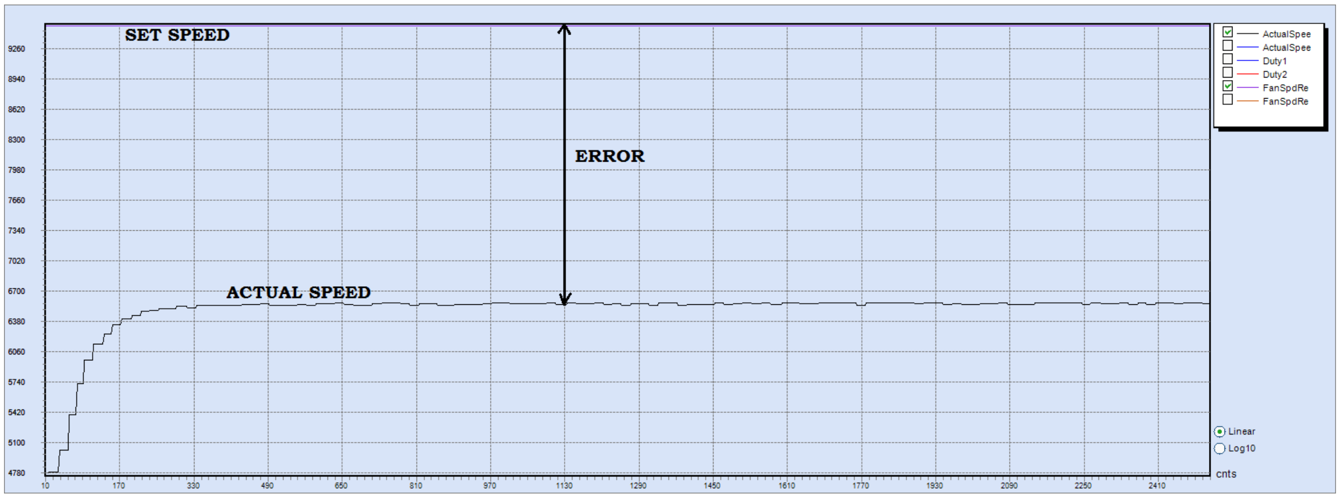
There is a tradeoff between a fast response and the stability factor. In a fan controller application, it is typically better to have a stable fan response as opposed to having a fast response time, because the temperature of a system does not change rapidly.

The PID TEST Command, provided in the *ClosedLoopFanControl.iic* file, generates a speed step input and reads the fan parameters. Prior to executing this command, first execute the Reset command that sets the fan speed to a default value of 4500 RPM. Then, execute the PID TEST command, which sets the fan speed to 9500 RPM, thereby generating a step input. Continuously execute this command by clicking the **Repeat** button, and go to the **Chart** window to observe the step response. Note that the PID TEST command currently updates FAN1, but you can change the fan identifier to generate a step input for FAN2.

Some example responses are given in Figure 48 to Figure 53. Notice the variations in steady state error, peak overshoot, rise time, and the settling time.

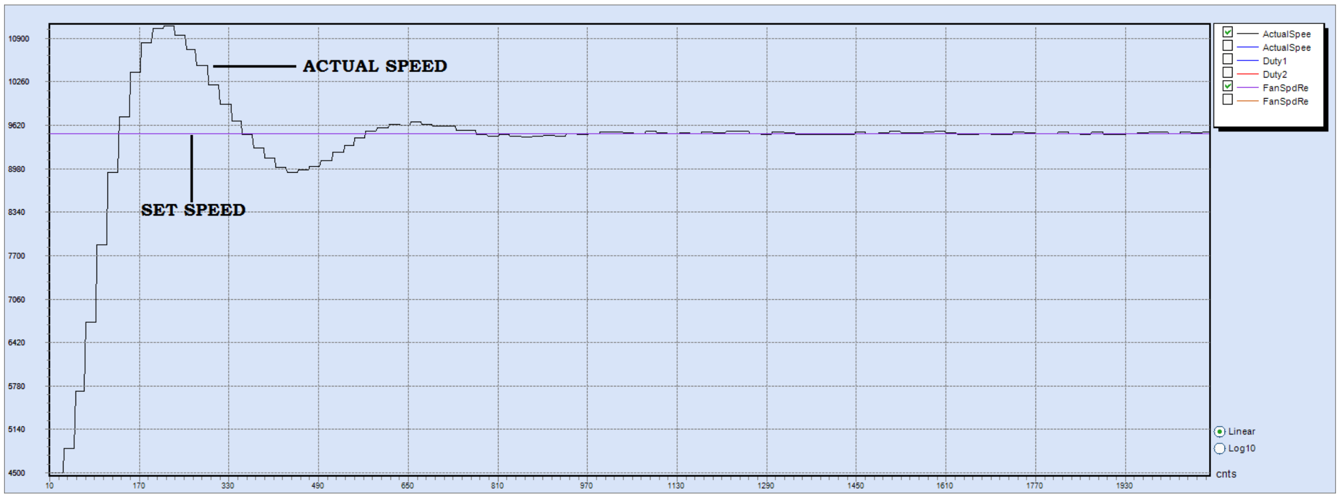
As Figure 48 shows, there is a huge error because of the absence of an integral control and the low value of the proportional parameter.

Figure 48. PID Tuning P = 30, I = 0, D=0



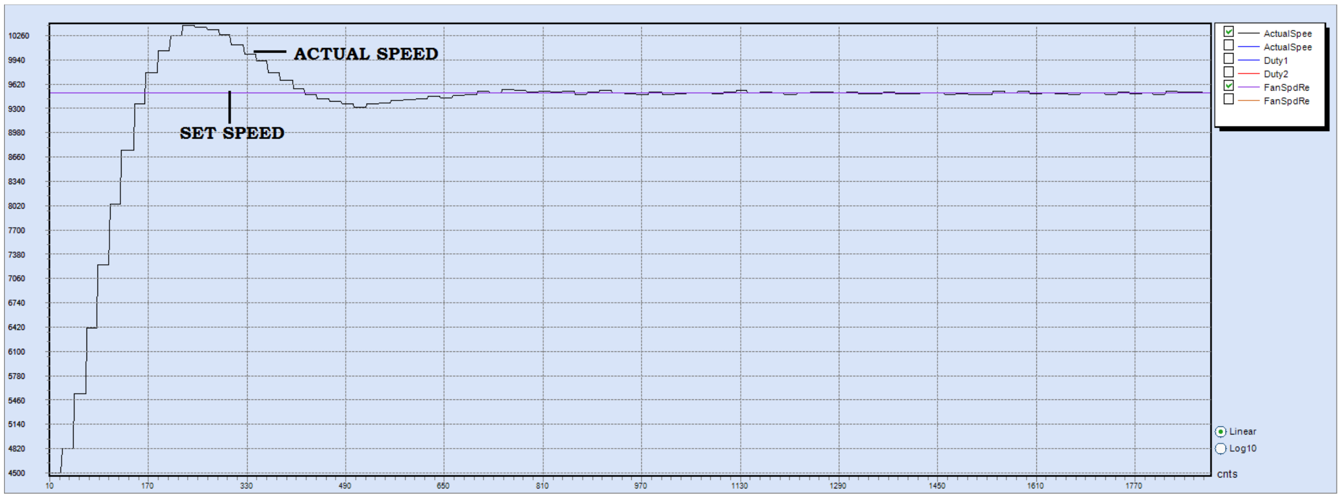
With the introduction of an integral control, the error is reduced to 0 but there is also a high overshoot, as Figure 49 shows.

Figure 49. PID Tuning P = 30, I = 30, D = 0



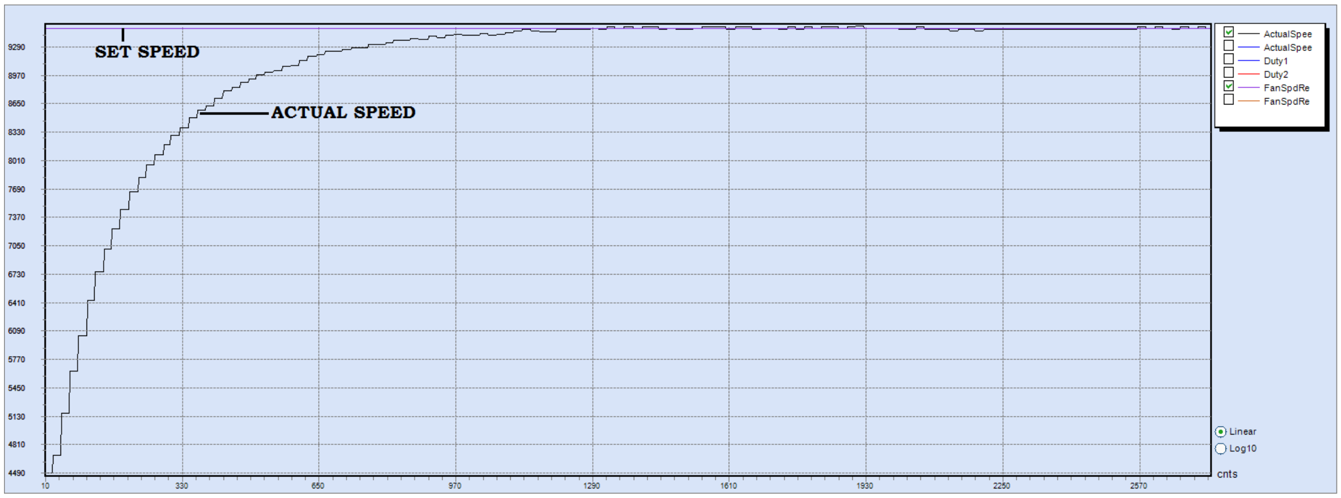
Reducing the integral parameter reduces the peak overshoot, as Figure 50 shows.

Figure 50. PID Tuning P = 30, I = 20, D = 0



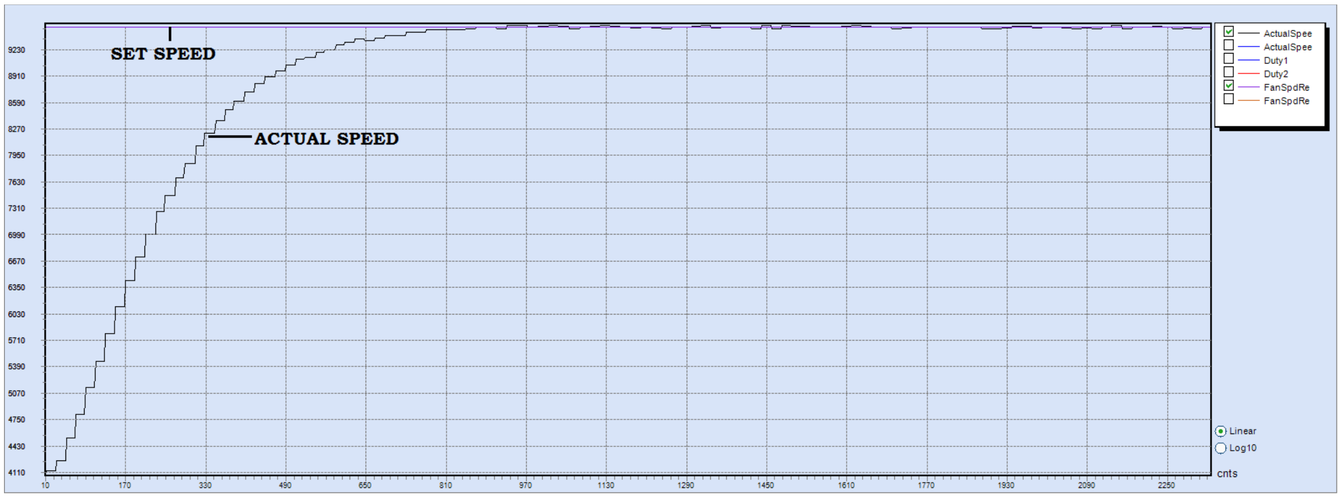
As shown in Figure 51, with further reduction of the integral parameter, peak overshoot becomes nil but the settling time increases.

Figure 51. PID Tuning P = 30, I = 5, D = 0



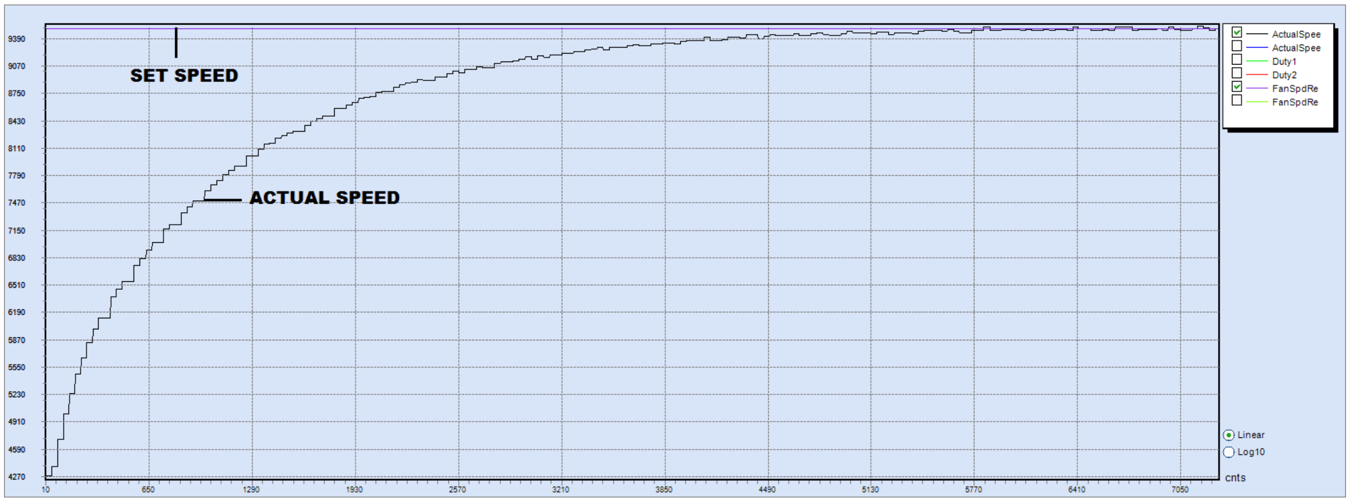
A reduction in the proportional parameter slows down speed response, as shown in Figure 52.

Figure 52. PID Tuning P = 15, I = 5, D = 0



A reduction in the integral parameter increases the settling time, as shown in Figure 53.

Figure 53. PID Tuning P = 15, I = 2, D = 0



Note When the PSoC 4 device is reset or power-cycled or being programmed, the PWM pins are tri-stated. This causes the fan to run at the maximum speed if the CY8CKIT-036 Thermal Management EBK is powered. If you want the fan to turn off during these conditions, connect the external pull-down resistors of around 4.7 KΩ to the PWM pins. The PWM pins are accessible on header J3 of the CY8CKIT-019 PSoC Shield Adapter.

Note When the CY8CKIT-036 Thermal Management EBK is powered after the PSoC 4 device, the fan starts spinning at high speed and gradually drops to the set value. This is because of the speed control loop running in the PSoC 4 device even before the Thermal Management EBK is powered. There are two ways to deal with this while developing the fan controller application:

1. Derive the voltage VDD for the PSoC 4 device from the same source, which drives the fan.
2. Detect the presence of the supply voltage to the fan and enable the fan controller.

Generally, this behavior will not arise as the controller and the fan derive power from the same source.

8 Project3_ThermalManagement

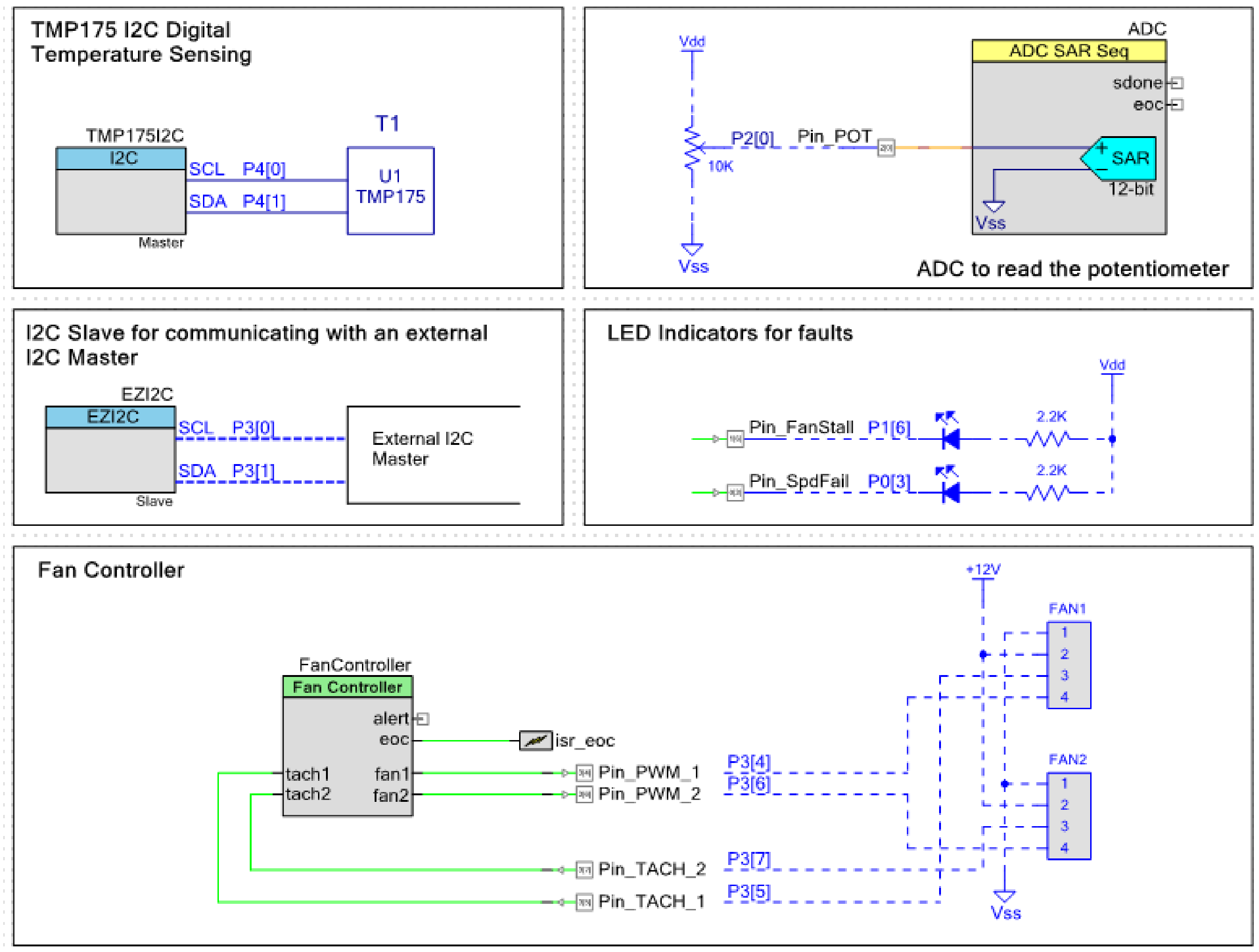
Close *main.c* and *TopDesign.cysch* for the previous project. Go to the Workspace Explorer window and right-click the project 'Project3_Thermal Management'. Select **Set As Active Project**.

This example project demonstrates a complete thermal management solution. It controls two fans based on inputs from two temperature measurement sources – an external I^2C digital temperature sensor and an analog output temperature sensor, emulated using a potentiometer. Note that in place of the potentiometer, a thermistor can be used with some additional code to calculate the temperature.

In a typical thermal management application, the system's physical area is split into multiple zones. Each zone has one or more fans and temperature sensors. In this project, two zones are defined using two sensors. The fan speed in a particular zone is calculated using an algorithm, which uses the output from both of the temperature sensors.

Figure 54 shows the TopDesign schematic of this project.

Figure 54. Project3_ThermalManagement - Top Design



8.1 Firmware Details

In this project, the FanController Component configuration is the same as in 'Project2_ClosedLoop'. Firmware is added to read the temperature from an external I^{2}C digital temperature sensor TMP175 and a potentiometer and adjust the fan speed.

Figure 55. Thermal Management - Block Diagram

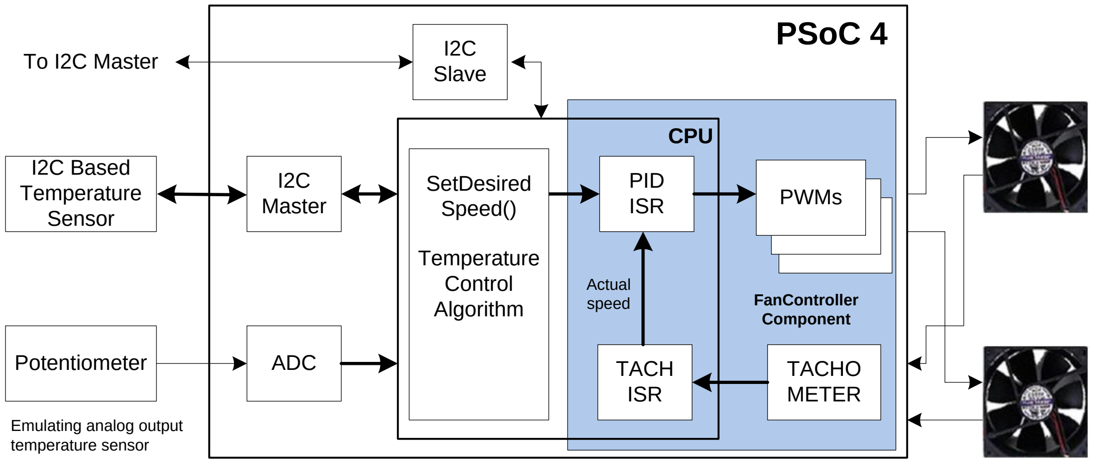
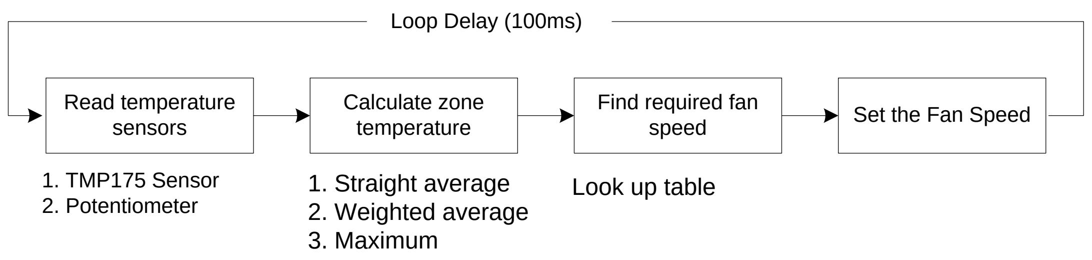


Figure 55 shows the block diagram of the project. The I^{2}C Master Component reads the temperature from an external I^{2}C digital temperature sensor. The 12-bit SAR ADC Component reads the potentiometer. Note that the potentiometer is used only to test the thermal management solution. In real applications, it is replaced by an analog sensor such as a thermistor. The I^{2}C Slave Component (EZI2C) is used for debugging.

In the project, the *ThermalManager.c* and *ThermalManager.h* files have APIs for developing a thermal management solution. Figure 56 shows the basic firmware flow.

Figure 56. Basic Firmware Flow



The SysTick interrupt from the Cortex M0 Processor is used to generate an interrupt every 100 ms. This controls the timing of the execution of the temperature measurement and control code. The temperature sensors are read at each interrupt. The temperature values are then used to calculate the zone temperature. As mentioned previously, two zones are defined in this project. The project firmware uses one of the following algorithms to combine the temperature sensor values and set the fan speed in a particular zone:

1. Straight average – calculates the simple average of the temperature sensor outputs in a zone with equal weights
2. Weighted average – calculate the weighted average of the temperature sensor outputs in a zone
3. Maximum – compares the temperature sensor outputs and picks the one which is highest value

For the zone configuration, a structure 'ZoneConfigEntry' is defined in *ThermalManager.c* to set parameters such as number of fans in the zone, required fan speed, number of temperature sensors, and weights for the sensors. As a default, both zones are configured to use both sensors with weighted average. For zone 1 controlled by FAN1, 90% weight is given to the potentiometer (T2) and 10% to the I^{2}C temperature sensor (T1). For zone 2 controlled by FAN2, 99% weight is given to the I^{2}C temperature sensor (T1) and 1% to the potentiometer (T2).

$$Zone\ 1\ Temperature, Z1 = \frac{1}{10}T1 + \frac{9}{10}T2$$
$$Zone\ 2\ Temperature, Z2 = \frac{99}{100}T1 + \frac{1}{100}T2$$

The calculated zone temperature is translated into a fan speed using a lookup table, ‘TemperatureToSpeedLookupTable’, defined in *ThermalManager.c.*

Figure 57 and Figure 58 show the graph of zone temperature versus fan speed for both thermal zones. Hysteresis is included to avoid fluctuations in the fan speed at the temperature set points. As a default, hysteresis is set to 4°C for zone 1 and 1°C for zone 2 in the structure ‘ZoneConfigEntry’.

Figure 57. Zone 1 Transfer Characteristics

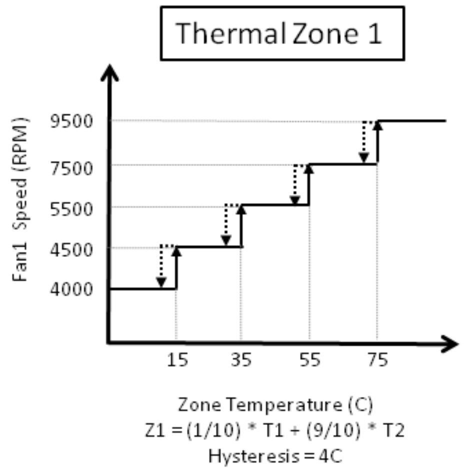


Figure 58. Zone 2 Transfer Characteristics

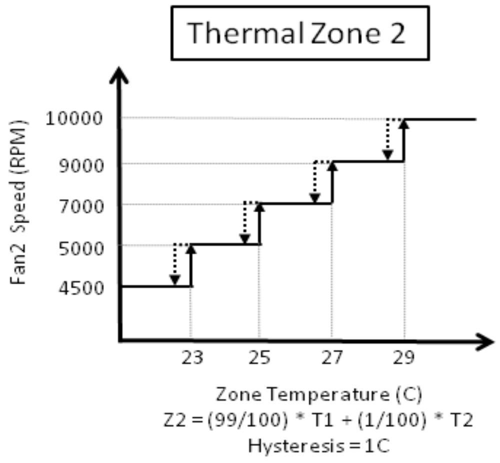
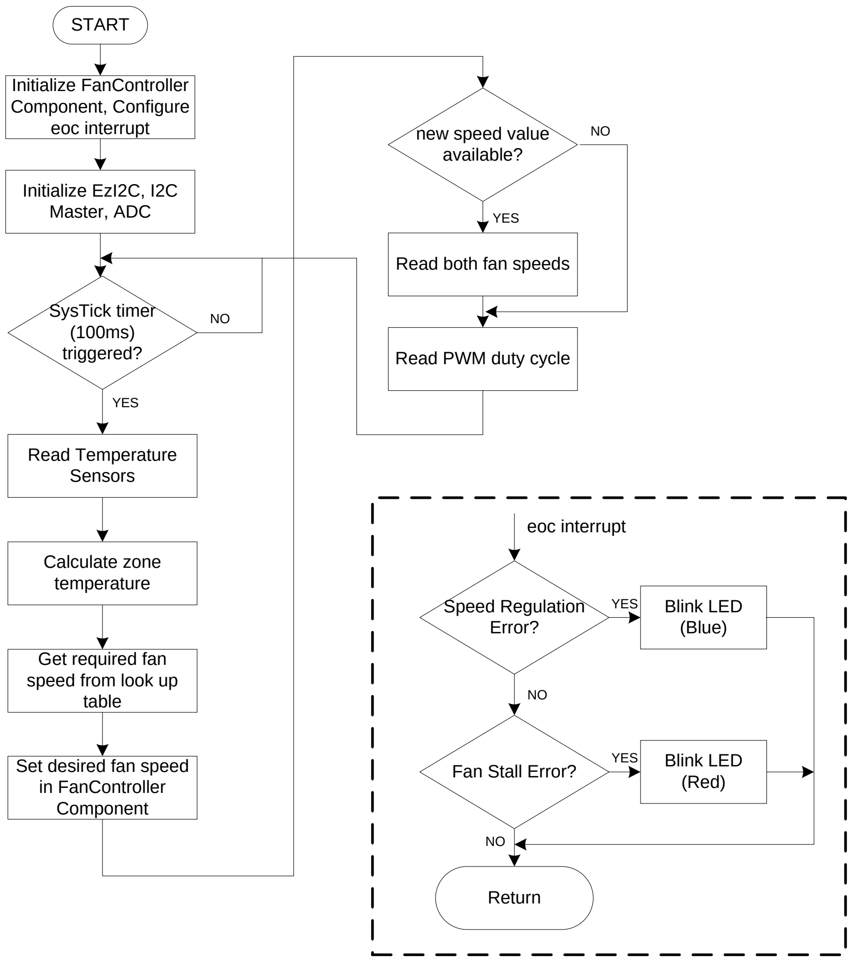


Figure 59 shows the detailed firmware flowchart.

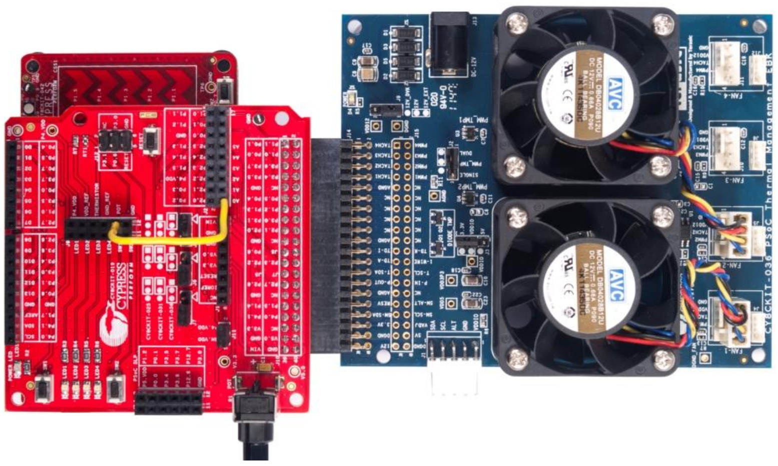
Figure 59. Firmware Flowchart


START
Initialize FanController Component, Configure eoc interrupt
Initialize EzI2C, I2C Master, ADC
SysTick timer (100ms) triggered?
NO
YES
Read Temperature Sensors
Calculate zone temperature
Get required fan speed from look up table
Set desired fan speed in FanController Component
new speed value available?
NO
YES
Read both fan speeds
Read PWM duty cycle
eoc interrupt
Speed Regulation Error?
YES
Blink LED (Blue)
NO
Fan Stall Error?
YES
Blink LED (Red)
NO
Return

8.2 Test Setup

Do the same Test Setup that was done in the first project, except the wire connection on the CY8CKIT-019 PSoC Shield Adapter. For this project, connect a wire between POT of header J9 to P2.0 of header J2 on the PSoC Shield Adapter, as Figure 60 shows. This is to connect the potentiometer to PSoC P2[0].

Figure 60. Complete Setup



8.3 Test procedure

For testing, use the BCP software. The following command script and variable initialization files are provided along with the project.

- *ThermalManagement.iic*
- *ThermalManagement.ini*

These files are in the zip file of the project provided with the application note. The file path is `/AN89346/BridgeFiles/`

This command script has a single read command to read the temperature values, set speed, actual speed and the duty cycle of both fans. Vary the potentiometer or temperature on the I^{2}C digital temperature sensor to see the fan speed variation according to Figure 57 and Figure 58, on page 36.

Note When the PSoC 4 device is reset or power-cycled or being programmed, the PWM pins are tri-stated. This causes the fan to run at the maximum speed if the CY8CKIT-036 Thermal Management EBK is powered. If you want the fan to turn off during these conditions, connect external pull-down resistors of around 4.7 KΩ to the PWM pins. The PWM pins are accessible on header J3 of the CY8CKIT-019 PSoC Shield Adapter.

Note When the CY8CKIT-036 Thermal Management EBK is powered after the PSoC 4 device, the fan starts spinning at high speed and gradually drops to the set value. This is because of the speed control loop running in the PSoC 4 device even before the Thermal Management EBK is powered. There are two ways to deal with this while developing the fan controller application:

1. Derive the voltage VDD for the PSoC 4 device from the same source, which drives the fan.
2. Detect the presence of the supply voltage to the fan and enable the fan controller.

Generally, this behavior will not arise as the controller and the fan derive power from the same source.

9 Summary

With the help of the FanController Component, you can quickly and easily create thermal management solutions with minimal firmware development.

The unique ability of the PSoC architecture to combine custom digital logic, analog functions, and an MCU in a single device lets you integrate many external fixed-function ASSPs. This powerful integration capability not only reduces BOM cost, but also results in PCB board layouts that are less congested and more reliable.

About the Author

Name:	Rajiv Vasanth Badiger
Title:	Application Engineer Staff
Background:	BE in Electronics and Communication from Nagpur University, India.

10 Related Application Notes

- AN66627 – PSoC 3 and PSoC 5LP Intelligent Fan Controller
- AN66477 – PSoC 3 and PSoC 5LP Temperature Measurement with a Thermistor
- AN70698 – PSoC 3 and PSoC 5LP Temperature Measurement with an RTD

Document History

Document Title: AN89346 - PSoC® 4 – Intelligent Fan Controller

Document Number: 001-89346

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	4208737	RJVB	12/03/2013	New application note.
*A	4599644	RJVB	12/19/2014	Updated Software Version as “PSoC Creator™ v3.0 SP2 or Higher” in page 1. Updated attached associated project to PSoC Creator 3.0 SP2 version.
*B	5537508	JOZH	12/14/2016	Sunset Review. Note that PSoC 4 mentioned in the article are PSoC 4200 parts; Update IDE version as “PSoC Creator™ v4.0 or Higher” in page 1; Refresh snapshots based on PSoC Creator^{TM} v4.0. Updated template
*C	5687870	AESATMP7	04/18/2017	Updated Cypress Logo and Copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709