AN66627

PSoC® 3 and PSoC 5 Intelligent Fan Controller

Author: Jason Konstas
Associated Project: Yes
Associated Part Family: PSoC 3, PSoC 5
Software Version: PSoC Creator™ v2.1

AN66627 demonstrates how to quickly and easily develop 4-wire brushless DC fan control systems using PSoC® 3 or PSoC 5. The five example projects included with this application note demonstrate how to use the supplied Fan Controller component to manage fans in a variety of common configurations. This application note also shows how to cut down development time for fan control systems from weeks to just a few hours.

Contents

Introduction

System cooling is a critical component of any high performance electronic system. As circuit miniaturization continues, increasing demands are placed on system designers to improve the efficiency of their thermal management designs.

Several factors conspire to make this a difficult task. Fan manufacturers specify duty-cycle to RPM relationships in their datasheets with tolerances as high as +/- 20%. To guarantee a fan will run at the desired speed, system designers would therefore need to run the fans at speeds 20% higher than nominal to ensure that any fan from that manufacturer will provide sufficient cooling. This results in excessive acoustic noise and higher power consumption.

Real-time closed loop control of fan speeds is possible using any standard microcontroller-based device running custom firmware algorithms, but this approach requires frequent CPU interrupts and constantly consumes processing power to achieve that basic, but fundamental task.

This application note shows how to use the FanController component in PSoC Creator to set up a thermal management system using PSoC 3 or PSoC 5. The first two examples expose designers to open-loop control mode (where the firmware is responsible for controlling fan speeds) and the associated application programming interfaces (APIs) that make that task quick and easy. The remaining three examples show how to configure the FanController into closed-loop control mode where the programmable logic resources in PSoC take care of fan control autonomously, freeing the CPU completely to do other important system management tasks.

4-Wire Fan Basics

A typical 4-wire brushless DC fan is shown in Figure 1. Two of the wires are used to supply power to the fan. The other two wires are used for speed control and monitoring.

Figure 1. Typical 4-Wire DC Fan

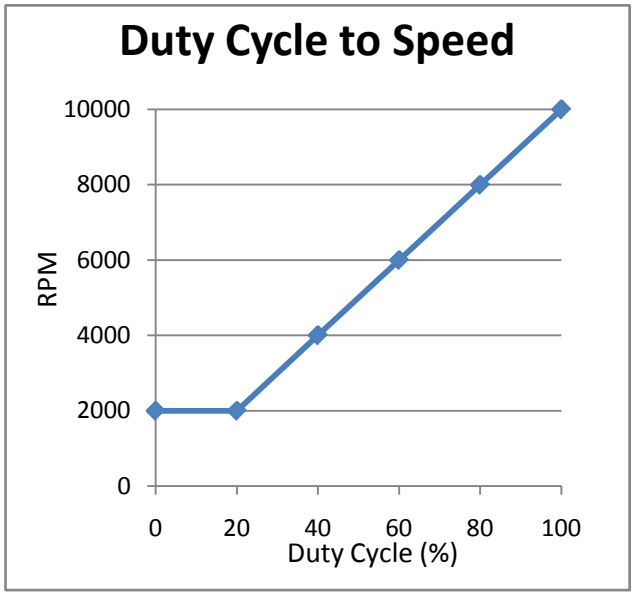
Fans come in standard sizes. 40 mm, 80 mm, and 120 mm are common. The most important specification when selecting a fan for a cooling application is how much air the fan can move. This is specified either as cubic feet per minute (CFM) or meters cubed per minute (m^3/min). The size, shape, and pitch of the fan blades all contribute to the fan's capability to move air. Obviously, smaller fans will need to run at a higher speed than larger fans to move the same volume of air in a given time frame. Applications that are space constrained and need smaller fans due to physical dimension limitations will generate significantly more acoustic noise. This is an unavoidable tradeoff that needs to be made to meet system level needs.

To manage acoustic noise generation, the FanController component can be configured to drive the fans at the minimum possible speed to maintain safe operating temperature limits. This also extends the operating life of the fan compared to systems that run all fans at full speed all of the time.

Fan Speed Control

With 4-wire fans, speed control is made possible through the use of a pulse-width modulator (PWM) control signal. Increasing the duty cycle of the PWM control signal will increase fan speed. Fan manufacturers specify how the PWM duty cycle relates to nominal fan speed. This is provided either through a table of data points or a graph that shows the relationship. An example of this information is shown below, with PWM control duty cycle in percentage shown on the horizontal axis and fan speed in RPM shown on the vertical axis.

Figure 2. Example Duty Cycle to Speed Chart



The FanController component provides a graphical user interface where designers can enter this information which then automatically configures and optimizes the firmware and hardware inside PSoC to control fans with these parameters.

It is important to note that at low duty cycles, 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. When entering duty cycle to RPM information into the FanController component customizer interface, select two data points from the linear region where the behavior of the fan is well-defined.

Measuring Fan Rotational Speeds

3-wire and 4-wire DC fans include hall-effect sensors that sense the rotating magnetic fields generated by the rotor as it spins. The output of the hall-effect sensor is a pulse-train that has a period inversely proportional to the rotational speed of the fan. The number of pulses that are produced per revolution depends on how many poles are used in the electromechanical construction of the fan.

For the most common 4-pole brushless DC fan, the tachometer output from the hall-effect sensor will generate two high and low pulses per fan revolution. If the fan stops rotating due to mechanical failure or other fault, the

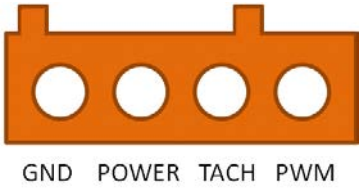
tachometer output signal will remain static at either a logic low or logic high level.

The FanController component measures the period of the tachometer pulse train for all fans in the system using a custom hardware implementation. The firmware APIs provided convert the measured tachometer periods into RPM to enable development of fan control algorithms that are firmware based. That same hardware block can generate alerts when it detects that a fan has stopped rotating, referred to as a stall event.

Fan Cables and Connectors

At the cabling level, wire color coding is not consistent across manufacturers, but the connector pin assignment is standardized. Figure 3 shows the connector pin out when viewed looking into the connector with the cable behind. Note the connectors are keyed to prevent incorrect insertion into the fan controller board. The keying scheme chosen also enables 4-wire fans to connect to control boards that were designed to support 3-wire fans (no PWM speed control signal) without modification.

Figure 3. 4-Wire DC Fan Connector Pin Assignment



The FanController component interacts with fans by driving the PWM speed control signals and monitoring fan actual speeds by measuring the tachometer (TACH) pulse trains. The component can be configured to automatically regulate fan speeds and detect fan failures based on the TACH inputs. Specifically how this is done will be presented in the example projects provided along with this application note.

Getting Started with the Fan Controller Component

Distributed with this application note is a PSoC Creator™ bundled project ZIP file that contains five example design projects and a library project that contains the **FanController** component. Save the ZIP file to a convenient location on your hard drive and extract the contents to a local folder.

To get started with the example projects, double-click the *AN66627.cywrk* PSoC Creator workspace file.

1. In the **Workspace Explorer** tab to the left of the screen, expand project **Example1** by clicking on the small '+' icon to the left of it.
2. Double-click *TopDesign.cysch* to open the top-level design schematic for the hardware blocks inside PSoC.

Figure 4 shows the top-level design schematic for Example Project 1.

All of the example projects provided with this application note are designed to run on the CY8CKIT-001 PSoC Development Kit (DVK), with support for two fans. The projects are easily modified to support more fans for further development and testing.

To replicate this environment properly, some wire connections are required on the CY8CKIT-001. Using the prototyping area on the DVK, wire power and ground to 4-pin headers for connecting to the fans. Using header **P19** on the DVK, make the six connections shown in blue color in Figure 4.

An example of this hardware setup is shown in Figure 5. The example shown uses two 12 V DC fans. Power for the fans is drawn directly from the DVK. It should be noted that the DVK can only provide around 1 A on the 12 V supply. If the fans you are working with exceed that current limit, use an external power supply to power the fans separately to avoid damaging the power electronics on the DVK.

Figure 4. Top-level Design Schematic for Example Project 1

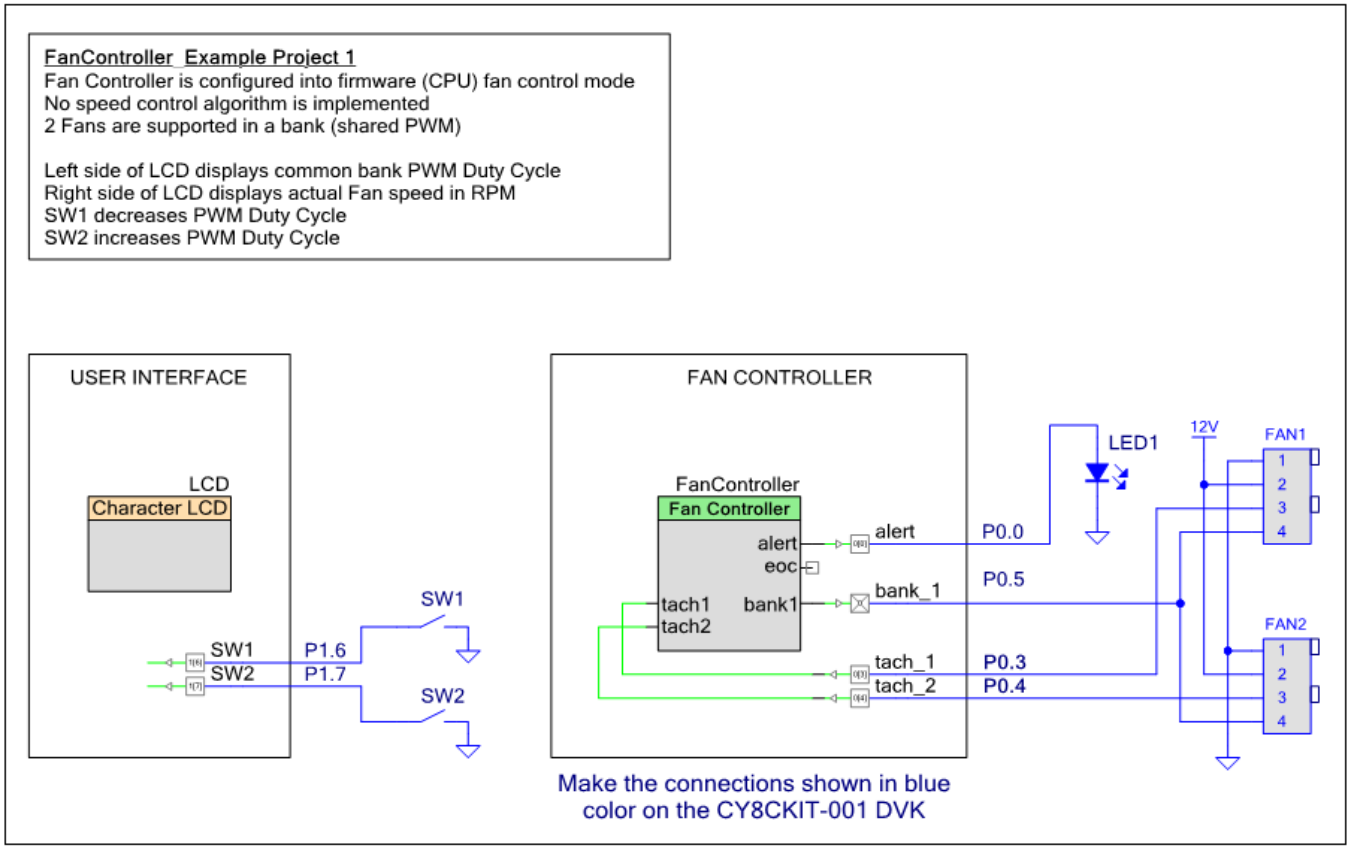
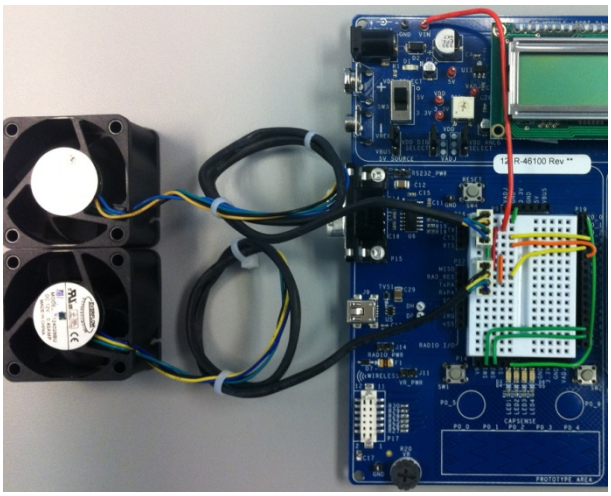
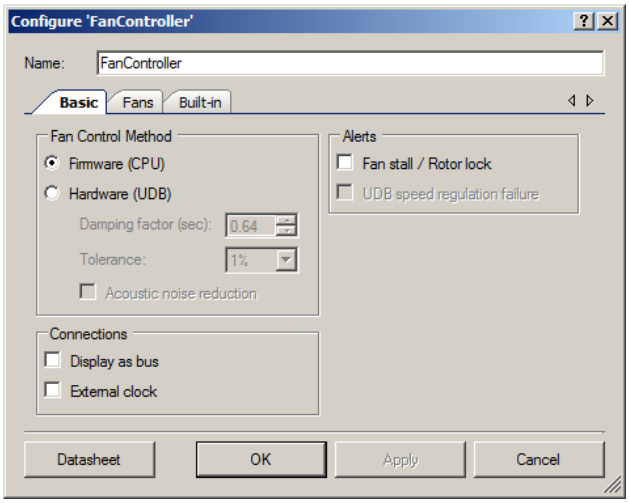


Figure 5. Wiring Fans to the DVK



Double-click the **FanController** component to open the component customizer as shown in Figure 6.

Figure 6. FanController Customizer – Basic Tab

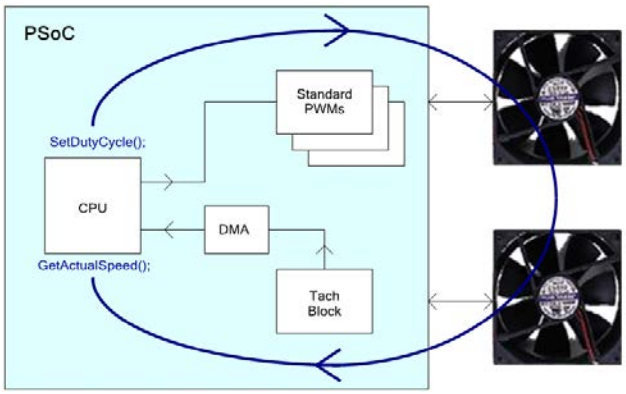


For the first example project, the FanController is configured to use firmware (CPU) fan control mode. In this mode, PSoC hardware blocks are used to implement the PWM blocks and the tachometer speed monitor (Tach) block, but firmware is entirely responsible for fan speed control.

A simplified representation of firmware fan control mode is shown in Figure 7. In this mode, firmware can set individual PWM duty cycles using the SetDutyCycle() API to change fan speeds. Actual speeds measured by the Tach block are transferred to SRAM using one channel of PSoC's DMA controller. Firmware can read individual fan speeds using the GetActualSpeed() API.

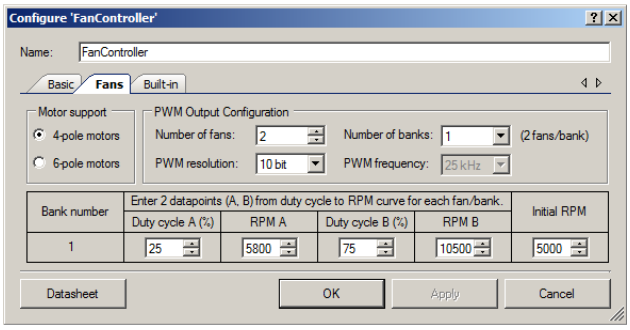
The Tach block can be configured to generate an alert signal if any of the fans stalls. This feature can be enabled by selecting the Fan Stall / Rotor Lock checkbox on the customizer Basic Tab, but leave that box unselected for the first example project.

Figure 7. Firmware Fan Control Mode



Move to the Fans Tab of the customizer to complete the configuration of the FanController component.

Figure 8. FanController Customizer – Fans Tab



This tab enables you to configure all of the parameters related to the PWM fan drivers. The number of fans including banking arrangement can be entered. A fan bank is defined as multiple fans that are driven by the same PWM speed control signal. For the first example, we will drive two fans with the same PWM signal representing one bank of fans.

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

The final section of this tab enables you to enter the electromechanical parameters of the fans you are working with. If you have this information from the fan manufacturer's datasheet, enter it here. The Duty A (%), RPM A, Duty 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 for the fans you are working with, stick with the default settings for now and you will be able to measure the actual duty cycle to speed relationship for your fans in the first example project. Once captured, you can come back into the customizer and enter those parameters later.

Now that the component configuration is complete, program the Example1 project into the DVK by selecting **Debug > Program** from the pull-down menus. Note that all the example projects will run on PSoC 3 or PSoC 5. The projects are configured for PSoC 3 by default. To switch to PSoC 5, simply change the target device using the **Device Selector** and rebuild the project to target the selected device.

If the project is running correctly, the text displayed on the debug LCD should display something like this:

```
50%    RPM1:  3250
Bank   RPM2:  3674
```

The same PWM duty cycle is being driven to both fans on PSoC port pin P0.5. Be sure to connect this signal to the PWM speed control pin on both fans for this example.

Example 1 – Firmware Details

In the **Workspace Explorer**, double-click *main.c* to examine the firmware used in this example project. The code is shown in code excerpt Code 1.

The purpose of this first example is twofold:

1) 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 same PWM duty cycle. This highlights that driving fans using conventional banks (a shared PWM drive signals across multiple fans) is far from optimal. In later examples when we explore closed loop hardware control mode, we will revisit this topic.

2) Capture duty cycle to RPM relationship if the fan manufacturer's datasheet is not available.

Looking at the code excerpt from example 1, it should be evident that there is no speed control algorithm implemented. Pressing switch SW1 on the DVK decreases the duty cycle by 5%. Doing so should result in a reduction in speed of both fans. Pressing switch SW2 on the DVK increases the duty cycle by 5%.

This example also introduces some of the basic APIs specific to the FanController component. Refer to the comments in the code excerpt to understand the relevance of the API calls and the component datasheet for a full list of APIs, their parameters, return values, and operation.

Code 1. Example1 *main.c* Firmware Listing

```
/* Duty cycles expressed in percent */
#define   MIN_DUTY    0
#define   MAX_DUTY    100
#define   INIT_DUTY   50
#define   DUTY_STEP   5

void main()
{
     uint16  dutyCycle = INIT_DUTY;

     /* Initialize the Fan Controller */
     FanController_Start();

     /* API uses Duty Cycles Expressed in
Hundredths of a Percent */
     FanController_SetDutyCycle(1,
dutyCycle*100);

     /* Initialize the LCD */
     LCD_Start();
     LCD_Position(0,0);
     LCD_PrintDecUint16(dutyCycle);
     LCD_PrintString("%");
     LCD_Position(1,0);
     LCD_PrintString("Bank");

     while(1)
     {
         /* Display Actual Speed Readings */
         LCD_Position(0,5);
         LCD_PrintString("RPM1: ");

LCD_PrintDecUint16(FanController_GetActualSpeed(
1));
         LCD_PrintString("     ");

         LCD_Position(1,5);
         LCD_PrintString("RPM2: ");

LCD_PrintDecUint16(FanController_GetActualSpeed(
2));
         LCD_PrintString("     ");
         CyDelay(250);

         /* Check for Button Press to Change
            Duty Cycle */
         if((!SW1_Read()) || (!SW2_Read()))
         {
             /* Increase Duty Cycle */
             if(!SW1_Read())
             {
                if(dutyCycle > MIN_DUTY)
                    dutyCycle -= DUTY_STEP;
             }

             /* SW2 = Increase Duty Cycle */
             else
             {
                if((dutyCycle += DUTY_STEP) >
MAX_DUTY)
                    dutyCycle = MAX_DUTY;
             }

             /* Adjust Duty Cycle of the Fan
                Bank */
             FanController_SetDutyCycle(1,
               dutyCycle*100);
             LCD_Position(0,0);
             LCD_PrintDecUint16(dutyCycle);
             LCD_PrintString("%  ");

             /* Switch Debounce *.
             CyDelay(250);
         }
     }
}
```

Example 2 – Firmware Speed Control

Close *main.c* and *TopDesign.cysch* for Example1. Go back to the Workspace Explorer and right-click on **Project 'Example2'.** Select **Set As Active Project**.

Open *TopDesign.cysch* for Example2 to start working on this project. The main change made to the top-level design schematic is the addition of a Status Register component available in the standard **Cypress Component Catalog** that the firmware can use to synchronize with hardware.

The purpose of this example is to use the same basic hardware configuration as Example1, but add firmware-based closed loop speed control. From example project 2 onward, the fans will have individual PWM control. Add a wire to the DVK to connect the second fan's PWM control input to PSoC port P0.6. This is shown in the top-level schematic for Example2.

Open the Component Customizer and go to the **Fans** tab. The one required change is to remove fan banking by setting the number of banks to 0. Doing so will expose an additional row to enter the electromechanical parameters for the second fan. Now is a good time to enter the fan parameters for your specific fans that you measured in example project 1. Note also that now that we are setting up the component to drive two fans independently, it is possible to enter different duty-cycle to RPM relationships for each fan. This capability allows the FanController component to work with any combination of fans, giving system designers more flexibility to mix and match different types of fans in their application to meet system cooling needs.

Now that the component configuration is complete, program the Example2 project into the DVK by selecting **Debug > Program** from the pull-down menus.

If the project is running correctly, the text displayed on the debug LCD should display something like this:

```
5000  5245  34.50%
F/W   4850  32.75%
```

The desired speed in RPM is displayed on the left. Pressing SW1 on the DVK decreases the desired speed by 500 RPM (SW2 increases by 500 RPM). A firmware algorithm responds by adjusting the duty cycle for both fans and continuously works at fine tuning the duty cycle until the actual fan speeds approach the desired speed.

The center of the display shows the actual speed in RPMs of both fans. If the actual speeds display zero, check the connection of the TACH signals from the fans. On the right of the display, the duty cycles for each fan are displayed for reference. In contrast to Example 1 where the fans were driven with the same duty cycle and we observed the difference in actual speeds, Example 2 shows the difference in duty cycles required to achieve the same desired speed on both fans.

The "F/W" text displayed on the bottom left of the display highlights that this project is using firmware speed control. This is done because the LCD display in example project 3 (when hardware closed loop control is introduced) is virtually identical, so this text helps to identify which example project is currently running on PSoC.

In the Workspace Explorer, double-click on *main.c* to examine the firmware used in this example project. The code is shown in code excerpt Code 2.

Code 2. Example2 *main.c* Firmware Listing

```
/* PWM duty cycle controls - units are hundredths of a percent */
#define MIN_DUTY            50
#define MAX_DUTY          9950
#define DUTY_STEP_COARSE   100
#define DUTY_STEP_FINE       2

/* Speed controls - units are RPM */
#define MIN_RPM           2500
#define MAX_RPM           9500
#define INIT_RPM          4500
#define RPM_STEP           500
#define RPM_DELTA_LARGE    500
#define RPM_TOLERANCE      100

void main()
{
     uint16  desiredSpeed = INIT_RPM;
     uint16  dutyCycle[2];
     uint8   fanNumber;
     char    displayString[6];

     FanController_Start();
     FanController_SetDesiredSpeed(1, desiredSpeed);
     FanController_SetDesiredSpeed(2, desiredSpeed);
     dutyCycle[0] = FanController_GetDutyCycle(1);
     dutyCycle[1] = FanController_GetDutyCycle(2);

     LCD_Start();
     LCD_Position(0,0);
     LCD_PrintDecUint16(desiredSpeed);
     LCD_Position(1,0);
     LCD_PrintString("F/W");

     while(1)
     {
          /* Synchronize firmware to end-of-cycle pulse from FanController */
          if (eocStatus_Read())
          {

               for(fanNumber = 1; fanNumber <= 2; fanNumber++)
               {
                    /* Display Fan Actual Speeds */
                    LCD_Position(fanNumber-1,5);
                    LCD_PrintDecUint16(FanController_GetActualSpeed(fanNumber));
                    LCD_PrintString("   ");

                    /* Firmware Speed Regulation */
                    LCD_Position(fanNumber-1,9);

                    /* Fan Below Desired Speed */
                   if(FanController_GetActualSpeed(fanNumber) < desiredSpeed)
                   {
                       if((desiredSpeed - FanController_GetActualSpeed(fanNumber)) > RPM_DELTA_LARGE)
                           dutyCycle[fanNumber-1] += DUTY_STEP_COARSE;
                       else
                           dutyCycle[fanNumber-1] += DUTY_STEP_FINE;
                       if(dutyCycle[fanNumber-1] > MAX_DUTY)
                       {
                           dutyCycle[fanNumber-1] = MAX_DUTY;
                       }
                   }

                   /* Fan Above Desired Speed */
                   else if(FanController_GetActualSpeed(fanNumber) > desiredSpeed)
                   {
                       if((FanController_GetActualSpeed(fanNumber) - desiredSpeed) > RPM_DELTA_LARGE)
```

```
                    {
                        if(dutyCycle[fanNumber-1] > (MIN_DUTY+DUTY_STEP_COARSE))
                            dutyCycle[fanNumber-1] -= DUTY_STEP_COARSE;
                    }
                    else if((FanController_GetActualSpeed(fanNumber) - desiredSpeed) > RPM_TOLERANCE)
                    {
                        if(dutyCycle[fanNumber-1] > MIN_DUTY)
                            dutyCycle[fanNumber-1] -= DUTY_STEP_FINE;
                    }
                }
                FanController_SetDutyCycle(fanNumber, dutyCycle[fanNumber-1]);

                /* Display Current Duty Cycle Settings (in 100ths of a percent) */
                LCD_Position(fanNumber-1,10);
                sprintf(displayString, "%5.2f", (((float)dutyCycle[fanNumber-1])/100));
                LCD_PrintString(displayString);
                LCD_PrintString("%    ");
            }
            CyDelay(250);

            /* Check for Button Press to Change Speed */
            if((!SW1_Read()) || (!SW2_Read()))
            {
                /* Decrease Speed */
                if(!SW1_Read())
                {
                    if(desiredSpeed > MIN_RPM)
                        desiredSpeed -= RPM_STEP;
                }

                /* Increase Speed */
                else
                {
                    desiredSpeed += RPM_STEP;
                    if(desiredSpeed > MAX_RPM)
                        desiredSpeed = MAX_RPM;
                }

                /* Display Updated Desired Speed */
                LCD_Position(0,0);
                LCD_PrintDecUint16(desiredSpeed);
                FanController_SetDesiredSpeed(1, desiredSpeed);
                FanController_SetDesiredSpeed(2, desiredSpeed);
                dutyCycle[0] = FanController_GetDutyCycle(1);
                dutyCycle[1] = FanController_GetDutyCycle(2);

                /* Switch Debounce */
                CyDelay(250);
            }
        }
    }
}
```

Because the time taken to measure the actual speed from each fan is long (between 12–18 ms per fan at 5000 RPM, for example), it is important that the firmware speed control algorithm knows when new actual speeds are available before making the next decision about how to adjust the PWM duty cycles. The end-of-cycle (**eoc**) output terminal on the FanController component serves this purpose. The **eoc** signal is generated by the Tach block inside the component and pulses high when a new actual speed sample is available from all fans connected to the component.

In this example project, the main firmware loop polls the status register component waiting for the end-of-cycle (**eoc**) pulse from the FanController component before running the duty cycle adjustment algorithm.

Example 3 – Hardware Speed Control

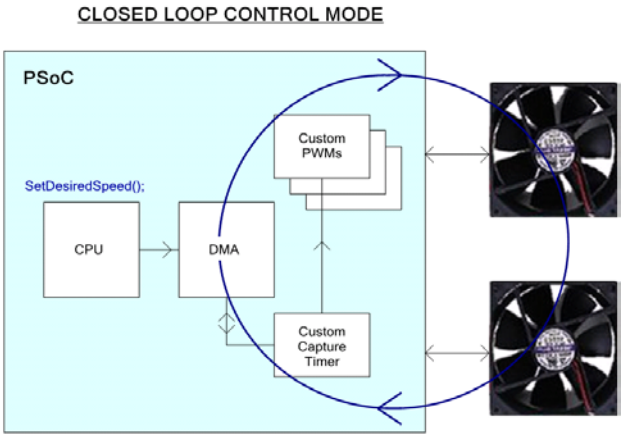
Close *main.c* and *TopDesign.cysch* for Example2. Go back to the Workspace Explorer and right-click on **Project 'Example3'.** Select **Set As Active Project**.

Open *TopDesign.cysch* for Example3 to start working on this project. The only change made to the top-level design schematic is the removal of the **eoc** Status Register component because firmware is not involved in the speed control in this project.

The purpose of this example is to replicate the functionality of example project 2 but this time the custom hardware blocks inside the FanController component perform the task of speed regulation, which frees the CPU to perform other tasks.

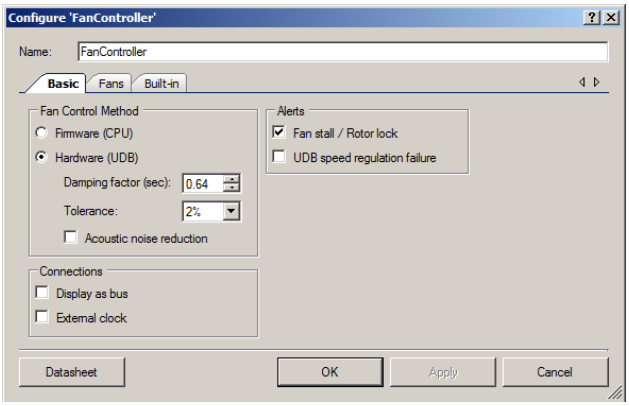
A simplified representation of hardware fan control mode is shown in Figure 9. In this mode, firmware can set desired fan speeds using the SetDesiredSpeed() API. The universal digital blocks (UDBs) inside the component will then work together to measure fan speeds and adjust duty cycles automatically. Actual fan speeds are still transferred to memory by the Tach block, which enables firmware to monitor fan speeds as a background activity if desired.

Figure 9. Hardware Fan Control Mode



Double-click the FanController component to open the component customizer as shown in Figure 10.

Figure 10. Configuring Closed Loop Control in the FanController Component Customizer



Hardware fan control mode is enabled by clicking the **Hardware (UDB)** radio button on the **Basic** tab. Doing so exposes some new controls that can be used to optimize the behavior of the closed loop hardware system.

Damping Factor

This parameter controls the dynamic response time of the hardware control loop. It controls how frequently the hardware will adjust the PWM duty cycles for each fan.

In situations where there are only a few fans, a higher damping factor will ensure that the control loop can regulate to the desired speed without oscillating around the desired speed target. In situations where many fans are being controlled, a lower damping factor will ensure adequate response time to changes in fan speed. This parameter enables fine tuning of the hardware control logic to match the selected fan's electromechanical characteristics.

The valid range for this parameter is 0..1.sec and the default setting is 640 ms. In example project 5, we will add an I^2C interface to the FanController component to enable real time monitoring of the fan speeds and the duty cycle adjustments made by the hardware control loop. That setup enables designers to experiment with different Damping Factor settings until the desired performance is achieved.

Tolerance

This parameter sets the acceptable tolerance when specifying desired fan speed targets. The tolerance is specified as a percentage relative to the desired speed setting. This parameter also enables fine tuning of the hardware control logic to match the selected fan's electromechanical characteristics.

The valid range for this parameter is 1..10%. Default setting is 1%. If 8-bit PWM resolution is selected in the FanController **Fans** tab, a Tolerance parameter setting of 5% is recommended.

Acoustic Noise Reduction

This parameter limits audible noise from fans by limiting the positive rate of change of speed. If enabled, and firmware requests an increase in desired fan speed, the PWM duty cycle applied to the fan will increase gradually to the new setting rather than applying a sudden step change. This eliminates noisy fan whir from sudden speed increases. Default setting is unselected. For this example project, leave this control unselected. We will enable it in later projects to see what effect this parameter has on the hardware control loop.

Alerts

For this example project, enable the Fan Stall / Rotor Lock alert. The alert pin is connected to LED1 on the DVK, so we have a visual indication if a fan stalls during operation.

Now that the component configuration is complete, program the Example3 project into the DVK by selecting **Debug > Program** from the pull-down menus.

If the project is running correctly, the text displayed on the debug LCD should display something like this:

```
4500  5245   34.50%
H/W   4850   32.75%
```

The desired speed in RPM is displayed on the left. Pressing SW1 on the DVK decreases the desired speed by 500 RPM (SW2 increases by 500 RPM). The hardware blocks in PSoC respond by adjusting the duty cycle for both fans and continuously work at fine-tuning the duty cycle until the actual fan speeds approach the desired speed.

In the Workspace Explorer, double-click on *main.c* to examine the firmware used in this example project. The code is shown in code excerpt Code 3.

This project has been configured to behave very similarly to example project 2 to demonstrate the power and convenience of hardware-driven closed loop control. The main loop in the firmware project deals only with the user interface—the switches for requesting changes to desired speed, the LCD to display fan information, and the LED to display alert status.

For testing purposes, a stall condition can be created by either forcing the fan to stop rotating or by removing the fan from the connector on the DVK. For safety reasons and to observe behavior in the case of a persistent fault, remove either one of the fan connections from the DVK. When the stall is detected, the LCD will display the word "STALL" in place of the actual speed reading and the red LED1 on the DVK will blink.

Code 3. Example3 *main.c* Firmware Listing

```
#define MIN_RPM   2500
#define MAX_RPM   9500
#define INIT_RPM  4500
#define RPM_STEP   500

void main()
{
    uint16  desiredSpeed = INIT_RPM;
    uint16  stallStatus;
    uint8   fanNumber;
    char    displayString[6];

    FanController_Start();
    FanController_SetDesiredSpeed(1, desiredSpeed);
    FanController_SetDesiredSpeed(2, desiredSpeed);

    LCD_Start();
    LCD_Position(0,0);
    LCD_PrintDecUint16(desiredSpeed);
    LCD_Position(1,0);
    LCD_PrintString("H/W");

    while(1)
    {
        /* Display Fan Actual Speeds and Duty Cycles */
        for(fanNumber=1; fanNumber<=2; fanNumber++)
        {
```

```
            LCD_Position(fanNumber-1,5);
            LCD_PrintDecUint16(FanController_GetActualSpeed(fanNumber));
            LCD_PrintString(" ");
            LCD_Position(fanNumber-1,10);
            sprintf(displayString, "%5.2f", (((float)FanController_GetDutyCycle(fanNumber))/100));
            LCD_PrintString(displayString);
            LCD_PrintString("%    ");

            /* Check for Fan Stall (poll alert status) */
            if(FanController_GetAlertSource())
            {
                stallStatus = FanController_GetFanStallStatus();
                if(stallStatus & fanNumber)
                {
                    LCD_Position(fanNumber-1,5);
                    LCD_PrintString("STALL");
                }
                CyDelay(250);
            }
        }

        /* Check for Button Press to Change Speed */
        if((!SW1_Read()) || (!SW2_Read()))
        {
            /* Decrease Speed */
            if(!SW1_Read())
            {
                if(desiredSpeed > MIN_RPM)
                    desiredSpeed -= RPM_STEP;
            }
           /* Increase Speed */
            else
            {
                desiredSpeed += RPM_STEP;
                if(desiredSpeed > MAX_RPM)
                    desiredSpeed = MAX_RPM;
            }
            FanController_SetDesiredSpeed(1, desiredSpeed);
            FanController_SetDesiredSpeed(2, desiredSpeed);

            /* Display Updated Desired Speed */
            LCD_Position(0,0);
            LCD_PrintDecUint16(desiredSpeed);

            /* Switch Debounce */
            CyDelay(250);
        }
    }
}
```

When an alert is generated, the firmware calls the GetFanStallStatus() API which de-asserts the alert pin. Because the stall condition is persistent, the alert is generated again the next time the Tach hardware block measures the fan speed. The 500 ms delay in the firmware loop ensures that LED1 stays lit long enough for you to see it.

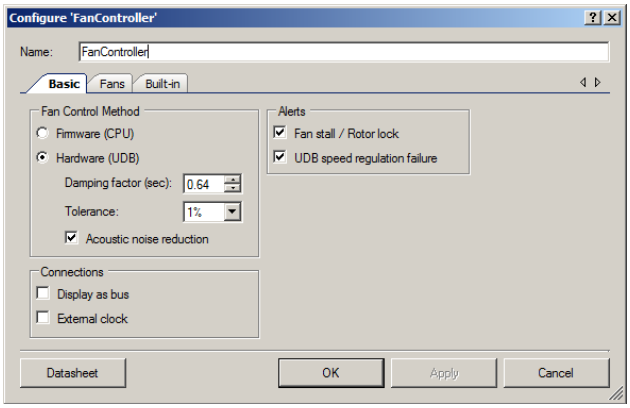
Example 4 – Advanced Hardware Speed Control

Close *main.c* and *TopDesign.cysch* for Example3. Go back to the Workspace Explorer and right-click on **Project 'Example4'.** Select **Set As Active Project**.

Open *TopDesign.cysch* for Example4 to start working on this project. The only change made to the top-level design schematic is the addition of an Interrupt component available in the standard **Cypress Component Catalog**.

Double-click the FanController component to open the component customizer as shown in Figure 11.

Figure 11. FanController Component Customizer Settings for Example Project 4



The purpose of this project is to demonstrate the Acoustic Noise Reduction feature of the hardware speed control logic and how to use interrupts to respond to alerts instead of polling alert status in firmware. Select the **Acoustic Noise Reduction** checkbox to enable that feature.

Speed Failure Alerts

Speed regulation failure alerts are generated by the closed loop hardware controller in two cases: 1) If the desired fan speed exceeds the current actual fan speed but the fan's duty cycle is already at 100%, 2) If the desired fan speed is below the current actual fan speed, but the fan's duty cycle is already at 0%.

For this example project, select the **Speed Failure** checkbox in the **Alerts** section of the **Basic** customizer tab.

Now that the component configuration is complete, program the Example4 project into the DVK by selecting **Debug > Program** from the pull-down menus.

If the project is running correctly, the text displayed on the debug LCD should be the same as that observed in example project 3.

If you left one or both of your fans disconnected from the DVK, you should see the word "STALL" shown on the LCD in place of the actual RPM. If so, this indicates that CPU interrupts are working and being serviced properly. We will examine the interrupt service routine code shortly. LED1 on the DVK is still connected to the alert pin, but now that the servicing of the alert is handled by interrupts instead of polling, the response time is significantly faster. You may not be able to see the alert pin toggling. Connect an oscilloscope or logic analyzer to the alert pin on your DVK to see the operation and timing of that signal.

You can confirm that Speed Regulation failure logic is working correctly in one of two ways:

1) Set a desired speed beyond the maximum speed that your fan can achieve. To do that, simply change the MAX_RPM #define in *main.c* and use SW2 to increase the desired speed to that setting. When speed regulation is not possible, the LCD displays "SPEED" in place of the actual RPM.

2) Set a desired speed below the minimum speed that your fan can achieve. To do that, simply change the MIN_RPM #define in *main.c* and use SW1 to decrease the desired speed to that setting. When speed regulation is not possible, the LCD will display "SPEED" in place of the actual RPM.

In the Workspace Explorer, double-click on *main.c* to examine the firmware used in this example project. The code is shown in code excerpt Code 4.

Because this project requires interrupts to function, the CYGlobalIntEnable macro is called during initialization to enable interrupts to the CPU core. The AlertInt_Start() API is provided with the interrupt component and needs to be called to have an interrupt vector and priority assigned to that interrupt source.

Code excerpt Code 5 shows the source code that needs to be manually entered into the API for the AlertInt component.

Near the top of the file, include the *FanController.h* file and reference the global variables defined in *main.c*. Then edit the AlertInt interrupt handler as shown.

The interrupt service routine for the alert signal calls the GetAlertSource() API to determine whether stall failures or speed regulation failures (or both) caused the alert. The GetFanStallStatus() or GetFanSpeedStatus() APIs return a bitmask with '1's populated in each bit corresponding to the faulty fan number(s). These APIs also de-assert the alert pin to prepare for future alerts.

As we saw during testing of this project, a persistent stall failure or speed regulation failure will generate interrupts continuously at a rate dictated by the amount of time taken for the Tach block to measure the actual speeds of all fans in the system. In that situation, the designer may choose to disable alerts from a known-faulty fan to prevent the continuous assertion of interrupts to the CPU core. This is achieved using the SetAlertMask() API which enables

masking of alerts from the specified fan number. Masking alerts from a fan disables generating of both stall and speed regulation alerts from that fan until further notice.

Code 4. Example4 *main.c* Firmware Listing

```
#define MIN_RPM    1000
#define MAX_RPM   20000
#define INIT_RPM   4500
#define RPM_STEP    500

uint16  stallStatus = 0;
uint16  speedStatus = 0;

void main()
{
    uint16 desiredSpeed = INIT_RPM;
    uint8  fanNumber;
    char   displayString[6];

    /* Globally Enable Interrupts to the CPU
Core */
    CYGlobalIntEnable;

    FanController_Start();
    FanController_SetDesiredSpeed(1,
desiredSpeed);
    FanController_SetDesiredSpeed(2,
desiredSpeed);
    FanController_SetDutyCycle(1, 1000);
    FanController_SetDutyCycle(2, 1000);
    AlertInt_Start();

    LCD_Start();
    LCD_Position(0,0);
    LCD_PrintDecUint16(desiredSpeed);
    LCD_Position(1,0);
    LCD_PrintString("H/W");

    while(1)
    {
        for(fanNumber=1; fanNumber<=2;
fanNumber++)
        {
            /* Check for Fan Stall and Speed
               Failure (flags set in AlertInt
               ISR) */
            LCD_Position(fanNumber-1,5);
            if(stallStatus & fanNumber)
            {
                LCD_PrintString("STALL");
                stallStatus &= ~fanNumber;
                CyDelay(500);
            }

            else if(speedStatus & fanNumber)
            {
                LCD_PrintString("SPEED");
                speedStatus &= ~fanNumber;
                CyDelay(500);
            }
            else
            {
                /* Display Fan Actual Speeds
                   When There are no Errors */

LCD_PrintDecUint16(FanController_GetActualSpeed(
fanNumber));
                LCD_PrintString(" ");
            }

            /* Always Display Fan Duty Cycles */
            LCD_Position(fanNumber-1,10);
            sprintf(displayString, "%5.2f",
              (((float)FanController_
                 GetDutyCycle(fanNumber))/100));
            LCD_PrintString(displayString);
            LCD_PrintString("%    ");
        }

        /* Check for Button Press to Change
           Speed */
        if((!SW1_Read()) || (!SW2_Read()))
        {
            /* Decrease Speed */
            if(!SW1_Read())
            {
                if(desiredSpeed > MIN_RPM)
                    desiredSpeed -= RPM_STEP;
            }

            /* Increase Speed */
            else
            {
                desiredSpeed += RPM_STEP;
                if(desiredSpeed > MAX_RPM)
                    desiredSpeed = MAX_RPM;
            }
            FanController_SetDesiredSpeed(1,
              desiredSpeed);
            FanController_SetDesiredSpeed(2,
              desiredSpeed);

            /* Display Updated Desired Speed */
            LCD_Position(0,0);
            LCD_PrintDecUint16(desiredSpeed);

            /* Switch Debounce */
            CyDelay(250);
        }
    }
}
```

Code 5. Example4 AlertInt.c Firmware Listing

```
/*****************************************
*****************************
*  Place your includes, defines and code here
*****************************************
*****************************/
/* `#START AlertInt_intc` */
#include "FanController.h"
extern uint16  stallStatus;
extern uint16  speedStatus;
/* `#END` */


CY_ISR(AlertInt_Interrupt)
{
     /*  Place your Interrupt code here. */
     /* `#START AlertInt_Interrupt` */

     uint8 alertStatus;

     /* Determine alert source: stall or speed
        regulation failure (could be both) */
     alertStatus = FanController_
                   GetAlertSource();

     /* If stall alert, determine which
        fan(s) */
     if (alertStatus & 0x01)
       StallStatus = FanController_
                     GetFanStallStatus();

     /* If hardware UDB speed regulation failure
        alert, determine which fan(s) */
     if (alertStatus & 0x02)
       SpeedStatus = FanController_
                     GetFanSpeedStatus();

    /* `#END` */
}
```

Example 5 – Real Time Monitoring

Close *main.c* and *TopDesign.cysch* for Example4. Go back to the Workspace Explorer and right-click on **Project 'Example5'.** Select **Set As Active Project**.

Open *TopDesign.cysch* for Example5 to start working on this project. The changes made to the top-level design schematic is the removal of the LCD and the user interface switches, replaced by the addition of the EzI2C Slave component available in the standard **Cypress Component Catalog**.

The purpose of this project is to enable you to monitor the operation of the FanController component in real time over I^2C using a graphical charting GUI. This is useful for evaluating the dynamic performance of the closed loop hardware controller and fine tuning and optimizing the parameters set in the component customizer.

In the Workspace Explorer, double-click on *main.c* to examine the firmware used in this example project. The code is shown in code excerpt Code 6.

This example highlights the benefit of using hardware closed loop fan control. There is zero code in the main firmware loop responsible for controlling the fans. The firmware is responsible only for handling the I^2C interface and translating I^2C data to commands for the FanController.

The example I^2C data structure enables an I^2C master to set desired speeds for each fan, read actual speeds, and current duty cycles.

A new API is introduced in this example: OverrideHardwareControl(). This API enables designers to take back control from the closed loop hardware and use firmware to control the fans. This could be useful in response to a fault condition or to implement a custom fan control algorithm temporarily. The same API can be used to pass fan control back to the hardware.

Program the Example5 project into the DVK by selecting **Debug > Program** from the pull-down menus. This project does not use the LCD, but it is not erased, so the LCD may still display data from Example4.

Code 6. Example5 *main.c* Firmware Listing

```
#define INIT_RPM          3000
#define NEW_COMMAND       0x0001
#define TOGGLE_OVERRIDE   0x8000

/* Define I2C Buffer Structure */
typedef struct
{
    /* R/W I2C variable – control word  */
    uint16 command;
    /* R/W I2C variables - desired speeds */
    uint16 desiredSpeed[2];
    /* R only I2C variables - actual speeds */
    uint16 actualSpeed[2];
    /* R only I2C variables - actual duty
       cycles*/
    uint16 dutyCycle[2];
} I2C_REGS;

void main()
{
    uint8 override = 0;

    /* Setup I2C Buffer */
    I2C_REGS i2cRegisters;
    I2C_Start();
    I2C_EnableInt();
    I2C_SetBuffer1(sizeof(i2cRegisters), 6 ,
                   (void *) &i2cRegisters);
    i2cRegisters.command = 0;
    i2cRegisters.desiredSpeed[0] = INIT_RPM;
    i2cRegisters.desiredSpeed[1] = INIT_RPM;

    /* Initialize Components */
    CYGlobalIntEnable;
    AlertInt_Start();
    FanController_Start();
```

```
    FanController_SetDesiredSpeed(1,
       i2cRegisters.desiredSpeed[0]);
    FanController_SetDesiredSpeed(2,
       i2cRegisters.desiredSpeed[1]);

    while(1)
    {
        /* Synchronize firmware to hardware
           end-of-cycle pulse */
        if(eocStatus_Read())
        {
            /* Check if I2C Host Sent New
              Desired Speeds */
            if(i2cRegisters.command &
               NEW_COMMAND)
            {
                FanController_SetDesiredSpeed(1,
                  i2cRegisters.desiredSpeed[0]);
                FanController_SetDesiredSpeed(2,
                  i2cRegisters.desiredSpeed[1]);
                i2cRegisters.command &=
                  ~NEW_COMMAND;
            }

            /* Check if I2C Host Sent Hardware
               Override Command */
            if(i2cRegisters.command &
               TOGGLE_OVERRIDE)
            {
                override ^= 1;
                FanController_Override
                  HardwareControl(override);
                i2cRegisters.command &=
                  ~TOGGLE_OVERRIDE;
            }

            /* Update I2C buffer with current
               Actual Speeds and Duty Cycles */
            i2cRegisters.actualSpeed[0] =
               FanController_GetActualSpeed(1);
            i2cRegisters.actualSpeed[1] =
               FanController_GetActualSpeed(2);
            i2cRegisters.dutyCycle[0]   =
               FanController_GetDutyCycle(1);
            i2cRegisters.dutyCycle[1]   =
               FanController_GetDutyCycle(2);
        }
    }
}
```

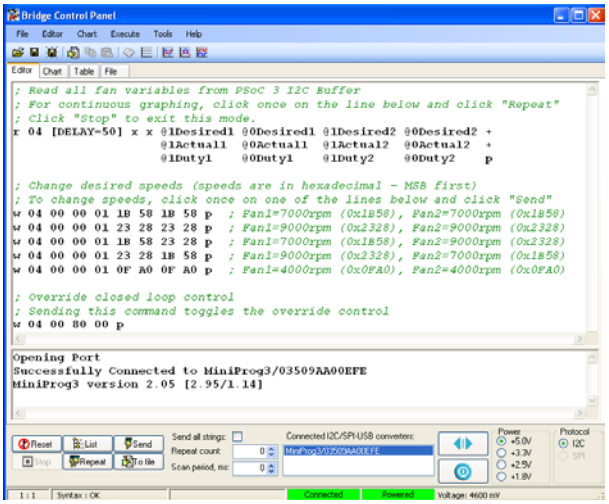
Using the USB-I²C Bridge Control Panel

The MiniProg3 programmer is also capable of implementing a USB-to-I²C bridge adaptor. We take advantage of this feature to send and receive commands to PSoC over I²C. Provided along with the example projects are a number of files that are intended for use with the Bridge Control Panel software. These are located in the folder called "**Bridge Control Panel Files**". To launch the Bridge Control software GUI, double-click on the relevant *.iic* file (**Fan_PSoC3.iic** or **Fan_PSoC5.iic**) in that folder. If the *.iic* file extension is not recognized on your computer, manually launch the Bridge Control Panel from the **Start** menu under the **Cypress** folder. Open the relevant *.iic* file manually if you started the software in that manner. Once open, select **Chart > Variable Settings** from the pull-down menu and click on **Load**. Open the *Fan.ini* file provided.

Figure 12 shows how the Bridge Control Panel GUI should look when launched and properly configured.

Leave the DVK powered on and remove the MiniProg3 from the PSoC processor module and connect the white 5-pin connector on the MiniProg3 to a 5-pin header you will need to wire up on the DVK as shown in the top-level schematic for this project. Ensure that you see the green "Connected" and "Powered" status boxes at the bottom of the Bridge Control panel GUI.

Figure 12. Bridge Control Panel GUI



The **Editor** tab is exposed by default whenever the Bridge Control Panel software is opened. The editor provides a simple text scripting capability to send and receive I²C data. For full details on the scripting language, refer to the Help provided in the USB-I²C Bridge Control Panel software.

The script provided for working with the Example5 project is shown in Code 7.

Code 7. USB-I²C Bridge Control Panel Script

```
; Read all fan variables from PSoC 3 I2C Buffer
; For continuous graphing, click once on the
; line below and click "Repeat"
; Click "Stop" to exit this mode.
r 04 [DELAY=50] x x @1Desired1 @0Desired1 +
                    @1Desired2 @0Desired2 +
                    @1Actual1  @0Actual1  +
                    @1Actual2  @0Actual2  +
                    @1Duty1    @0Duty1    +
                    @1Duty2    @0Duty2    p

; Change desired speeds (speeds are in
; hexadecimal - MSB first)
; To change speeds, click once on one of the
; lines below and click "Send"

; Fan1=7000rpm (0x1B58), Fan2=7000rpm (0x1B58)
w 04 00 00 01 1B 58 1B 58 p
; Fan1=9000rpm (0x2328), Fan2=9000rpm (0x2328)
w 04 00 00 01 23 28 23 28 p
; Fan1=7000rpm (0x1B58), Fan2=9000rpm (0x2328)
w 04 00 00 01 1B 58 23 28 p
; Fan1=9000rpm (0x2328), Fan2=7000rpm (0x1B58)
w 04 00 00 01 23 28 1B 58 p
; Fan1=4000rpm (0x0FA0), Fan2=4000rpm (0x0FA0)
w 04 00 00 01 0F A0 0F A0 p

; Override closed loop control
; Sending this command toggles the override
; control
w 04 00 80 00 p
```

The format used in the script language is simple. The "r" at the beginning of a script line indicates an I²C read transfer. The next byte (in hexadecimal) is the I²C address in 7-bit format.

After the address is an optional delay parameter, specified in milliseconds. This is useful for setting a display update rate when continuous graphing is enabled.

The script line then contains a list of variable names. These have been set to match the I²C data structure defined in Example5. The "x x" indicates that the MiniProg3 should read the first 2 bytes from PSoC and discard them. Those first 2 bytes are used to send commands to PSoC, so we don't need to read or view them here. We are interested in viewing the desired speeds, actual speeds and duty cycles from the FanController. Those are all 16-bit values transported over the native 8-bit I²C bus. The "@1" and "@0" prefixes to the variable names define the endian ordering of the bytes so that when we view the data in the chart GUI, the numbers will appear as 16-bit values.

Finally, the "p" at the end of each script line directs the MiniProg3 to generate an I²C stop condition terminating the transfer.

Using this script file, we are able to accomplish several things:

1. Read FanController speeds and duty cycles in real time by continuously executing the first script line.

2. Set new desired speeds as shown in the second section of the script file as outlined in the comments.

3. Toggle between hardware closed loop control mode and firmware control mode as shown in the third section of the script file.

The first exercise to gain familiarity with the Bridge Control Panel utility will be to display the actual speeds in real time and graph them over time. The power on default speed for both fans was set at 3,000 RPM in *main.c*.

Click once on the script line beginning with "r" and then click on the **Repeat** button at the bottom of the GUI. When you see the status text scrolling in the window at the bottom, click on the **Chart** tab to see a display of the results.

Figure 13 shows an example of a fan speeding up from stall with a low Damping Factor parameter set. The red line shows the target speed of 3000 RPM. The green line shows the actual speed ramping up from zero, overshooting the target, and then oscillating a few times around the target speed before converging. The blue line shows the duty cycle controlled by the closed loop controller hardware. The positive and negative duty cycle adjustments are evident.

Figure 14 shows an example of a fan speeding up from stall with a medium Damping Factor parameter set. The green line shows the actual speed ramping up from zero, overshooting once but then quickly settling back to the desired speed within tolerance.

Figure 15 shows an example of a fan speeding up from stall with a high Damping Factor parameter set. The green line shows the actual speed ramping up very slowly from zero, achieving the target speed and settling immediately. Feel free to experiment with various combinations of the closed loop hardware parameters. Don't be alarmed if you initially find that the closed loop controller cannot achieve regulation and the actual speed oscillates forever around the target desired speed. This is expected when a small number of fans are connected and a very low Damping Factor is set and/or a low tolerance setting. Some tuning will be required to find the optimal set of control loop parameters.

Figure 13. Closed Loop Fan Control – Speed up from Stall (Low Damping Factor)

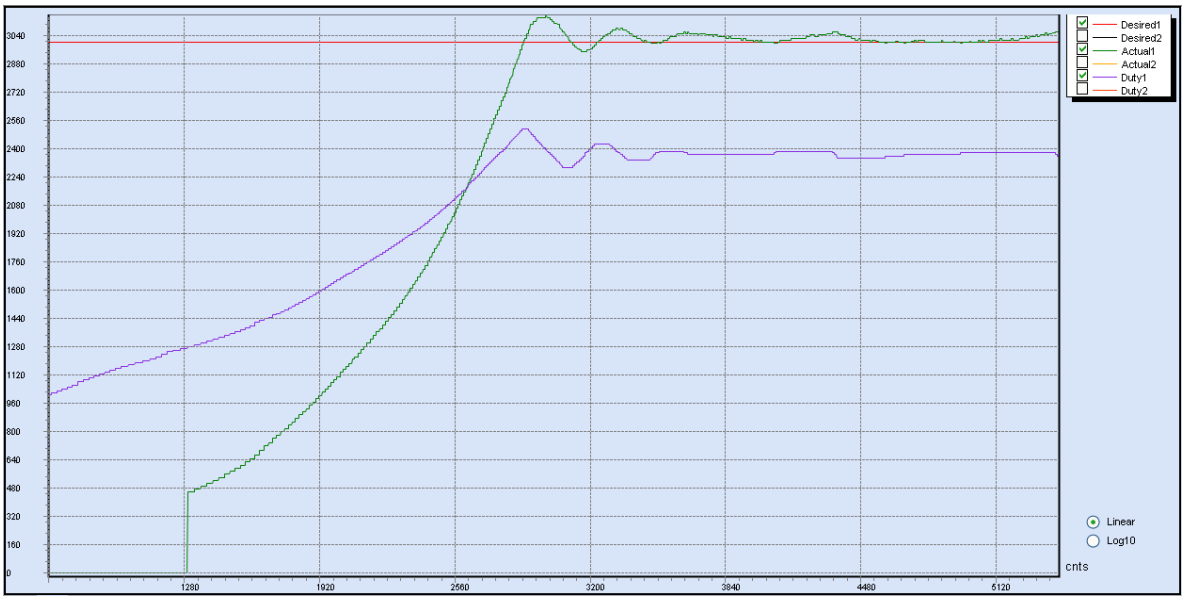


Figure 14. Closed Loop Fan Control – Speed up from Stall (Medium Damping Factor)

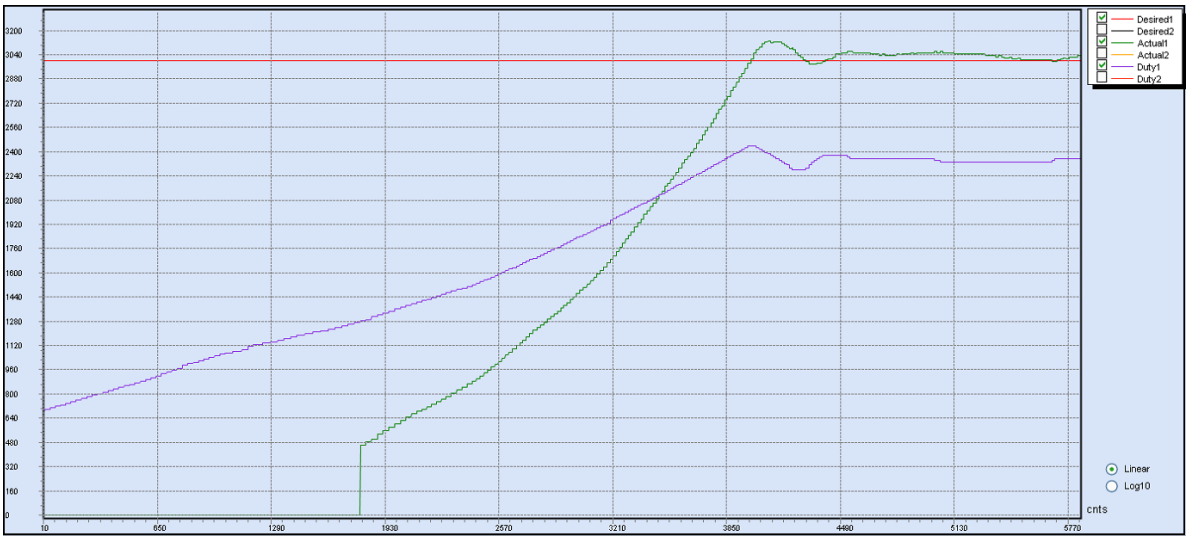


Figure 15. Closed Loop Fan Control – Speed up from Stall (High Damping Factor)

The I^2C write commands in the script enable you to independently change the desired speed for each fan. You can use these commands to see how the FanController responds to speed changes and verify the dynamic behavior of the control loop. To send a command, click once on the script line and click on the **Send** button or simply press Enter.

Remember that when Acoustic Noise Reduction is enabled, the FanController component limits the positive rate of change in fan speed. Changing desired fan speed from 2,000 to 10,000 RPM results in a gradual speed change, far more pleasing to the ear than a sudden speed change that not only generates annoying fan whirr noise, but also causes sudden current surges in the fan's power supplies. Experiment with this feature by enabling or disabling it in the **Basic** tab of the component customizer.

The I^2C script file provided also enables you to turn closed loop control mode off by toggling the override control to see the difference in fan behavior when it is running in open loop mode compared to closed loop control mode. Disturbing the fan's airflow by either blocking the air intake or blowing air against the flow causes noticeable changes in fan speed in open loop control mode. When closed loop control is enabled, the FanController will respond quickly and return the fans to their desired speed within tolerance.

PSoC Thermal Management EBK

If you would like to continue working with the FanController component, consider purchasing the PSoC Thermal Management Expansion Board Kit (CY8CKIT-036) shown below in Figure16.

The kit comes with two fans installed plus a variety of analog and digital temperature sensors to enable you to quickly prototype a complete thermal management system.

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

Figure16. CY8CKIT-036 PSoC Thermal Management Expansion Board Kit (EBK)

Conclusion

If you have worked through all the example projects, you will have gained a good understanding of the various operating modes of the FanController component and the associated APIs. The component enables designers to quickly and easily begin working on thermal management applications with minimal firmware development required.

Closed loop hardware control mode provides many benefits including reduced firmware development time and optimal fan speeds reducing acoustic noise and power consumption. The custom logic available within the PSoC architecture provides immense value in these applications, enabling designers to build complex control systems in hardware, freeing up the CPU to handle other tasks such as protocol handling or other critical system management tasks.

The unique ability of the PSoC architecture to combine custom digital logic, analog signal chain processing and an MCU in a single device enables system designers to 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.

Document History

Document Title: PSoC® 3 and PSoC 5 Intelligent Fan Controller – AN66627

Document Number: 001-66627

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	3159398	JK	2/10/2011	New Application Note – Initial Release
*A	3265851	JK	6/30/2011	Changed application note title. Updated the example projects to use Fan Controller Component v1.10
*B	3550058	JK	3/13/2012	Updated document to new Application Note template format. Updated the example projects for use with PSoC Creator v2.0. Removed distribution of FanController component library. Added section introducing the CY8CKIT-036 PSoC Thermal Management Expansion Board Kit (EBK)
*C	3576608	JK	4/10/2012	Updated the example projects to use the new FanController v1.21 component to address a defect found in the SetDesiredSpeed() API Updated template according to current Cypress standards.
*D	3676155	JK	07/13/2012	Updated the example projects to use the production FanController component v2.10. Updated screenshots of configuration GUI for the FanController component v2.10

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

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

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

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com