(FALL 2021) EE128 Mini Project

Smart Fan Control

Tyler Chargualaf

Po-Han Wang

Project Summary

This system outputs a PWM wave used as an input signal to a PWM controlled fan. The automatic fan control works by translating the ambient temperature to an appropriate fan speed (Duty Cycle for the PWM). The system also features a button for cycling between 3 manual speed settings and the automatic setting. Unlike the manual settings, the automatic fan control features a power saving mode that "turns off" the output signal after 10 seconds of no motion (easily configurable for up to 7.8 hours).

Functions

- Onboard **SW2** interrupt for Mode Cycling (auto, speed 1, speed 2, speed 3)
- 25KHz PWM output
- ADC of Temperature Sensor translated to Linear Fan curve
- Power Saving Motion Detector

Parts

- K64F
- MCP9700A-E/TO Temperature Sensor
- PIR SENSOR IR WIDE ANGLE
- Green LED
- 1k Resistor
- 330 Resistor
- Jumper Wires
- 2 x 0.1uF Ceramic Capacitors
- PWM controlled Fan

Port Connections (5 physical wires out of K64F)

- ADC_DP0 <= Temperature Sensor
- PTD0 <= PIR Sensor OUT
- A1 => Green LED
- PTB22 => Onboard Red LED
- PTC6 <= Onboard SW2
- 3V3 => Breadboard Power rail
- GND => Breadboard Power rail

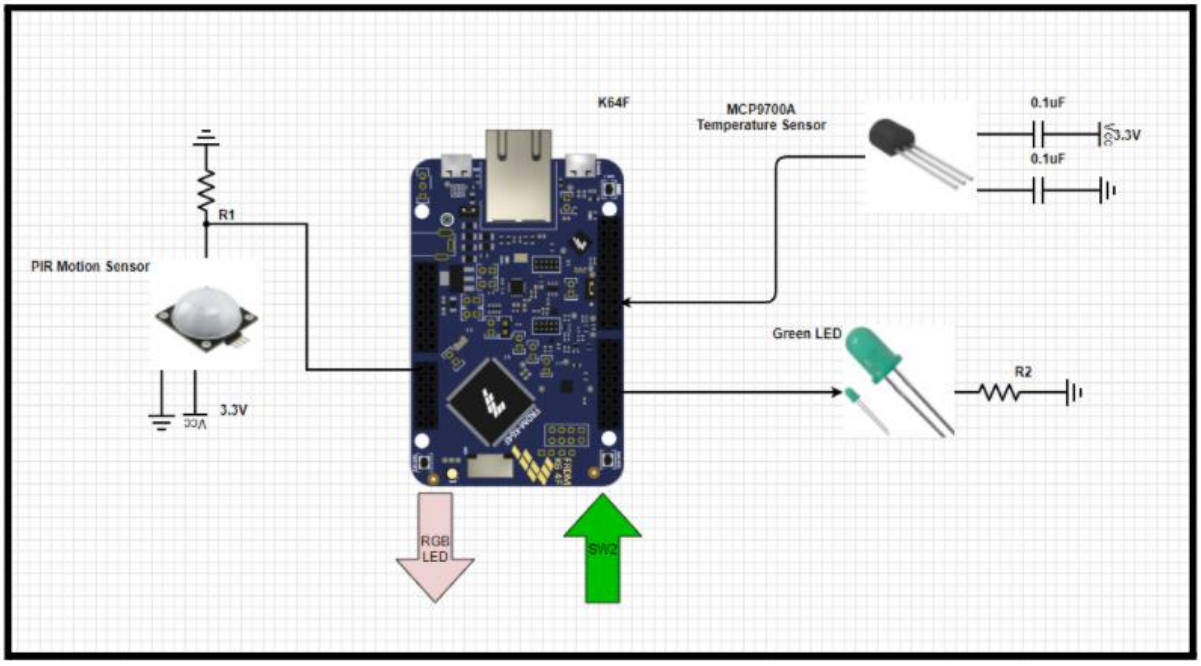
Important Note

We do not have the PWM Fan that this system would work on. After discovering an issue with the DC fan, we ordered, we modified the project to work on a PWM Controlled Fan. The system would fully function in providing a PWM input signal to control the fan speed.

Because of this, the PWM output was instead connected to an LED. This model actually makes it easy to visualize the PWM Duty Cycle changing with respect to inputs.

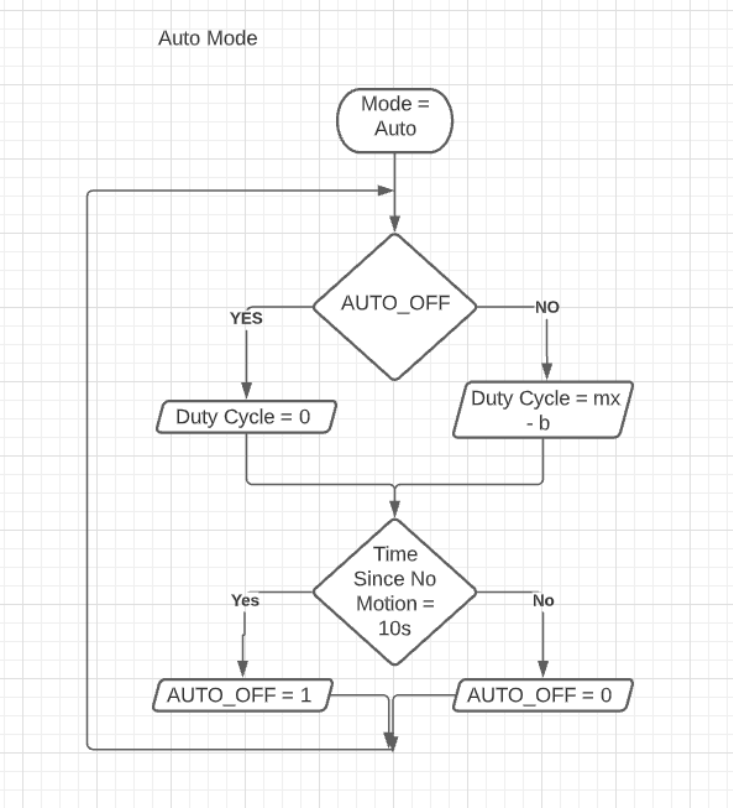
System Design

Block Diagram

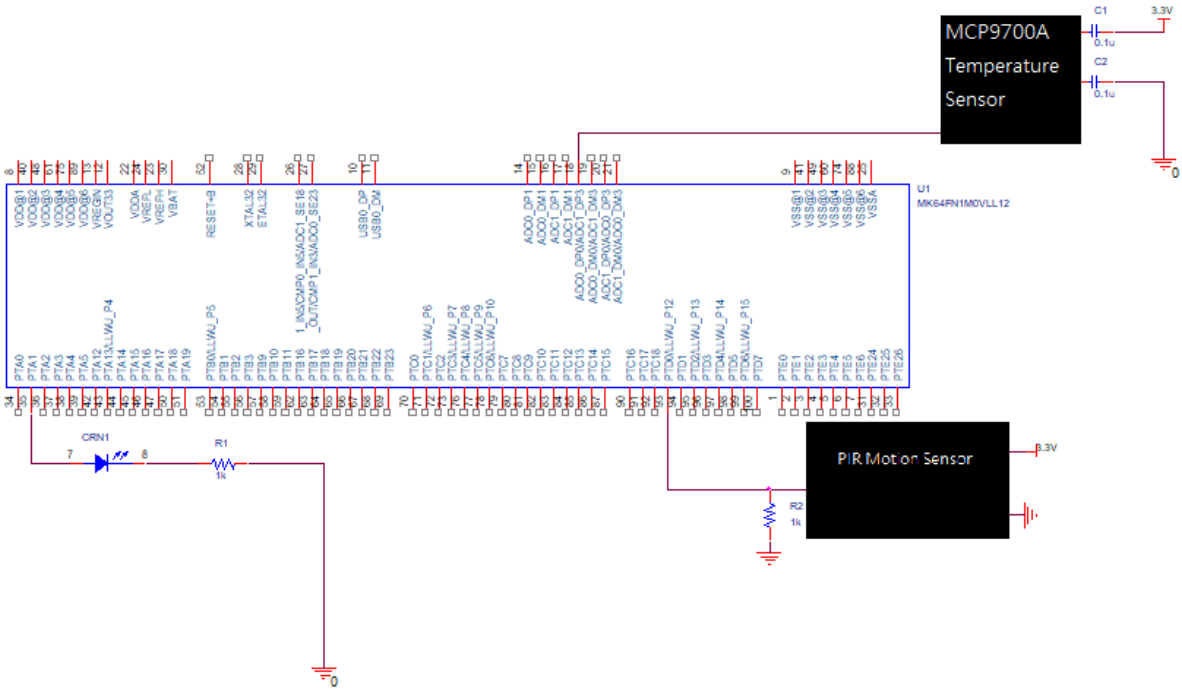


Simple Visual Diagram (3.3V and GND are connected to K64F)

Flow Chart



Implementation



Note: In this schematic is an LED used to visualize the PWM Duty Cycle. The PWM output for this system works on a selected PWM Fan.

Main Loop

```
for (;;) {
    //-----------------------------------------------------
    //Read Temperature
    adc_val = (((3.3)* ADC_read16b()) / 65536 ) - 0.5;
    adc_val = (adc_val) / 0.01;
    float temperature = adc_val;
    //-----------------------------------------------------
    //Check Mode and Calculate speed
    unsigned char temp_mode = mode % 4;
    if ((temp_mode == 0) && (AUTO_OFF == 0)) {
        //3.0 (Duty/Celsius) Linear Fan Curve
        //0% at 4C, 100% at 37.3C

        //FAN CURVE
        new_speed = (unsigned short)((((adc_val * 3) - 12) / 100.0) * 879); //0.00 to 1.00

        //TEST
        //new_speed = (adc_val > 27.0) ? 879 : 0;


        if (new_speed > 879) new_speed = 879;   //100%
    }
    else if (temp_mode == 0 && AUTO_OFF == 1) new_speed = 0; // 0%
    else if (temp_mode == 1) new_speed = 263;   //30%
    else if (temp_mode == 2) new_speed = 527;   //60%
    else if (temp_mode == 3) new_speed = 791;   //90%
    //-----------------------------------------------------
    //Control PWM Duty
    if (FTM0_CNT <= FTM0_C6V)
        updatePWM(new_speed);
    else if (FTM0_C6V == 0 && new_speed != 0)
        updatePWM(new_speed);

    //-----------------------------------------------------
}
```

Timer

```
void FTM3_IRQHandler(void) {
    nr_overflows++;
    if (GPIOD_PDIR & 0x01) {
        nr_overflows = 0;
        AUTO_OFF = 0;
    }
    else if (nr_overflows >= 25) {
        AUTO_OFF = 1;
    }
    uint32_t SC_VAL = FTM3_SC;
    FTM3_SC &= 0x7F; // clear TOF
}
```

Functions

```
static inline void updatePWM(unsigned short speed)
{
    FTM0_SC &= 0x8000;  //Disable FTM TIMER
    FTM0_C6V = speed;   //Change pulse width
    FTM0_SC |= 0x8; //Write 01 to CLKS to enable FTM TIMER
}

unsigned short ADC_read16b(void)
{
    ADC0_SC1A = 0x0; //Write to SC1A to start conversion from ADC_0
    while(ADC0_SC2 & ADC_SC2_ADACT_MASK); // Conversion in progress
    while(!(ADC0_SC1A & ADC_SC1_COCO_MASK)); // Until conversion complete
    return ADC0_RA;
}
```

Switch Interrupt Handler

```
void PORTC_IRQHandler(void)
{
    if (mode % 4 == 0) GPIOB_PCOR |= (1 << 22);          /*Turn on  Red Led*/
    else if (mode % 4 == 3) GPIOB_PSOR |= (1 << 22);     /*Turn off Red Led*/
    mode += 1;                                           /*Cycle Mode*/
    PORTC_ISFR = PORT_ISFR_ISF(0x40);                    /* Clear interrupt status flag */
}
```

Testing

The procedure for testing varied between checking values at breakpoints and observing the real time response to input. LEDs were useful in testing.

- **Temperature**
    - Calculated Ambient temperature from ADC was within +- 0.5 of the ambient temperature measured by a Home Thermostat
    - Temperature sensor was heated using a hairdryer
        - **Software Verification**
            1. Record Temperature value at room temperature and Duty Cycle
            2. Breakpoint
            3. Heat Temperature sensor with hairdryer
            4. Resume until after ADC is read
            5. Record Temperature value and increased Duty Cycle
        - **Hardware**
            1. Cycle to Automatic
            2. Observe Brightness of Green LED
            3. Heat Sensor with hairdryer
            4. Observe Brightness of LED increasing to nearly brightness of 3.3V supply
- **PWM**
    - Preferred testing environment would be with an oscilloscope
    - Fan testing was not feasible
        - Fan too small
        - Hard to visualize fan speed
    - **Tested with Green LED**
        1. Cycle through manual modes
        2. Observe brightness increasing (speed 1 to 2 to 3)
        3. Cycle to Automatic
        4. Observe Brightness
        5. Heat Temperature Sensor
        6. See Brightness increase
- **Motion Sensor**
    1. Set sensitivity of sensor low
    2. Cycle mode to Automatic
    3. Run hand in front of sensor
    4. Sensor stays high for 3 seconds after motion detected (Indicated by LED on Sensor)
    5. Wait 10 seconds after Sensor LED turns off
    6. Green LED turns off
    7. Repeat
- **Onboard Switch**
    1. Cycle through modes
    2. See Green LED Brightness changing as well as RED LED indicator for Manual Modes

Discussions

Challenges

- Originally expected to drive a DC fan with PWM signal
    - DC Fan speed changes by varying DC voltage across the rated voltage
    - PWM did not work correctly even for high frequencies
        - Stalling? Inertia?
    - Changed PWM frequency to be used as an INPUT signal instead of Power
- ADC reading not accurate to ambient temperature
    - Bypass capacitor at V+
    - Experimented with different reference voltages and formulas for translating ADC reading to Temperature
- Changing PWM cycle without pausing execution
    - Changed FTMx_CHnV while FTMx_CNT > FTMX_CHnV
        - Changing Pulse Width while signal is low so signal is not kept high/low unintentionally

Limitations

- Testing Temperature Sensor
    - ADC value increased accordingly, but accuracy of temperature reading past room temperature was unknown
- 25KHz PWM may not work with all PWM Fans
    - Must be changed in code
- Temperature Sensor location
    - Close to MCU

Possible Improvements

- Temperature sensor calibration
    - Resources provided by manufacturers
        - High complexity
- Piecewise Fan Curve
    - Fan curves are not usually single lines
        - Exponential curves towards high temperatures
- Floating Point arithmetic not needed
    - Done for simplicity and testing
    - ADC readings can be directly translated into respective Duty Cycle
        - Lookup table
- User Settings
    - Change Fan Curve (duty cycle / degree temperature)
    - Adjust Power-Saving time
        - Many ways to do this
        - Could involve using a display
            - Too costly
- Remote Use

Roles and Responsibilities

Tyler Chargualaf:

- Code
- Flowchart
- Circuit assembly and testing
- Discussion
- Part Selection

Po-Han Wang:

- Discussion
- Schematics
- Electrical circuit verification
- Code review
- Part review

Design decisions/considerations were communicated, discussed, and implemented remotely. Tyler received a Stay-At-Home notice and was not able to meet with partner in person to work on project.

Conclusion

This project was a good exercise in designing a real-time system. Of course, many of the design decisions were done with respect to a relatively stable testing environment without consideration for extreme environments. Many parts which can be used in such projects have varying levels of complexity depending on the designer's purposes. All in all, it was helpful practice in referencing datasheets, manufacturer resources, manuals, and more.

Video Demo https://youtu.be/BLRwBStb1T0

Description: For Testing Purposes, Sensitivity of Motion Sensor was set to lowest possible setting. Normally, it is very sensitive. Still, the hairdryer interfered with the readings as is shown in the video. Because it is difficult to demonstrate the PWM with this interference, the difficulty of steadily and slowly heating up the temperature sensor, and the LED brightness not being picked up very well by the camera, the demo uses a cutoff instead of a linear fan curve for the automatic temperature-controlled setting. It can be seen in the source code that the system works with the linear fan curve and not the cutoff.

```c
#include "fsl_device_registers.h"
unsigned int nr_overflows = 0;
unsigned short ADC_read16b(void);
unsigned char mode = 0;
unsigned char AUTO_OFF = 0;

static inline void updatePWM(unsigned short speed);

int main(void)
]{
    //Variables Initialization
    float adc_val = 0;
    unsigned short new_speed = 0;
    //Clock Gating
    SIM_SCGC5 |= SIM_SCGC5_PORTA_MASK;      /*Enable Port A Clock Gate Control*/
    SIM_SCGC5 |= SIM_SCGC5_PORTD_MASK;      /*Enable Port D Clock Gate Control*/
    SIM_SCGC5 |= SIM_SCGC5_PORTC_MASK;      /*Enable Port C Clock Gate Control*/
    SIM_SCGC5 |= SIM_SCGC5_PORTB_MASK;      /*Enable Port B Clock Gate Control*/
    SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK;       /* 0x8000000u; Enable ADC0 Clock*/
    SIM_SCGC6 |= SIM_SCGC6_FTM0_MASK;   /*Enable the FTM0 Clock*/

    SIM_SCGC3 |= SIM_SCGC3_FTM3_MASK;   /*Enable the FTM3 Clock*/



    //IO Configuration
    PORTD_PCR0  = 0x0100;       /*Configure D0 as GPIO*/
    PORTB_PCR22 = 0x100;        /*Red Led, configured as Alternative 1 (GPIO)*/
    GPIOB_PDDR |= (1 << 22);    /*Setting the bit 21 of the port B as Output*/
    PORTA_PCR1 = PORT_PCR_MUX(3);       /*MUX = ALT 4*/
    GPIOD_PDDR &= 0x0;          /*Configure D0 as input*/

    //PWM Configuration
    //25KHz     Interrupt Enabled   Active HIGH (Channel match LOW) CLKS = 00
    FTM0_SC &= 0x8000;          /*TOIE[6]CLKS[4:3]PS[2:0]*/
    FTM0_MOD = 879;             /*Setting the Modulo register*/
    FTM0_C6SC = 0x0028;         /*Setting MSB = 1, ELSnB = 1*/
    FTM0_C6V = 0;               /*Value of the Channel*/

    //Motion Sensor Polling
    //0.4s Counter period
    //10s when nr_overflows = 25
    FTM3_SC |= 0x004F;  /*TOIE = 0 CPWMS = 0 CLKS = 01; PS = 111*/
    FTM3_MOD = 0xFFFF;  /*FTM Clock Period*/

    //Interrupt Configuration
    PORTC_PCR6  = 0x90100;      /*PORTC_PCR6: ISF=0,IRQC=9,MUX=1*/
    PORTA_PCR4  = 0x100;        /*Changing the NMI to GPIO*/

    //ADC Configuration
    ADC0_CFG1 = 0x0C;           /*Configure ADC for 16 bits, and to use bus clock*/
    ADC0_SC1A = 0x1F;           /*Disable the module, ADCH = 11111*/

    PORTC_ISFR = PORT_ISFR_ISF(0x40);     /* Clear interrupt status flag */

    NVIC_EnableIRQ(PORTC_IRQn);         /*Enable the PORTC interrupt*/
    NVIC_EnableIRQ(FTM3_IRQn);          /*Enable FTM3 interrupt*/
```

```
    for (;;) {

        //-----------------------------------------------------
        //Read Temperature
        adc_val = (((3.3)* ADC_read16b()) / 65536 ) - 0.5;
        adc_val = (adc_val) / 0.01;
        float temperature = adc_val;
        //-----------------------------------------------------
        //Check Mode and Calculate speed
        unsigned char temp_mode = mode % 4;
        if ((temp_mode == 0) && (AUTO_OFF == 0)) {
            //3.0 (Duty/Celsius) Linear Fan Curve
            //0% at 4C, 100% at 37.3C

            //FAN CURVE
            new_speed = (unsigned short)((((adc_val * 3) - 12) / 100.0) * 879); //0.00 to 1.00

            //TEST
            //new_speed = (adc_val > 27.0) ? 879 : 0;


            if (new_speed > 879) new_speed = 879;   //100%
        }
        else if (temp_mode == 0 && AUTO_OFF == 1) new_speed = 0; // 0%
        else if (temp_mode == 1) new_speed = 263;   //30%
        else if (temp_mode == 2) new_speed = 527;   //60%
        else if (temp_mode == 3) new_speed = 791;   //90%
        //-----------------------------------------------------
        //Control PWM Duty
        if (FTM0_CNT <= FTM0_C6V)
            updatePWM(new_speed);
        else if (FTM0_C6V == 0 && new_speed != 0)
            updatePWM(new_speed);

        //-----------------------------------------------------

    }

    return 0;
}
void FTM3_IRQHandler(void) {
    nr_overflows++;
    if (GPIOD_PDIR & 0x01) {
        nr_overflows = 0;
        AUTO_OFF = 0;
    }
    else if (nr_overflows >= 25) {
        AUTO_OFF = 1;
    }
    uint32_t SC_VAL = FTM3_SC;
    FTM3_SC &= 0x7F; // clear TOF
}A

static inline void updatePWM(unsigned short speed)
{
    FTM0_SC &= 0x8000;  //Disable FTM TIMER
    FTM0_C6V = speed;   //Change pulse width
    FTM0_SC |= 0x8; //Write 01 to CLKS to enable FTM TIMER
}

unsigned short ADC_read16b(void)
{
    ADC0_SC1A = 0x0; //Write to SC1A to start conversion from ADC_0
    while(ADC0_SC2 & ADC_SC2_ADACT_MASK); // Conversion in progress
    while(!(ADC0_SC1A & ADC_SC1_COCO_MASK)); // Until conversion complete
    return ADC0_RA;
}

void PORTC_IRQHandler(void)
{
    if (mode % 4 == 0) GPIOB_PCOR |= (1 << 22);         /*Turn on  Red Led*/
    else if (mode % 4 == 3) GPIOB_PSOR |= (1 << 22);    /*Turn off Red Led*/
    mode += 1;                                          /*Cycle Mode*/
    PORTC_ISFR = PORT_ISFR_ISF(0x40);                   /* Clear interrupt status flag */
}
```