Motor Controller With Current Feedback

Jeremy Carleton
Sefayet Saikot

Project Description

The objective for this project was to use the FRDM board to control a motor using feedback from a current sensor such that the motor draws a certain amount of current specified by the user. The current sensor has an analog output, and the control signal for the motor is a PWM signal sent to the electronic speed controller. The FRDM board receives the desired current draw from the computer that it is connected to over a UART link, specified as a number between 0 and 800 where 0 corresponds to no current and 800 is the limit for the amount of current that the controller can reliably track. It also outputs a stream of data allowing the current measured and the PWM duty cycle to be monitored by the user. A switch is connected to one of the GPIO pins to act as an emergency brake by forcing the PWM duty cycle down to 19%, causing the motor to stop.

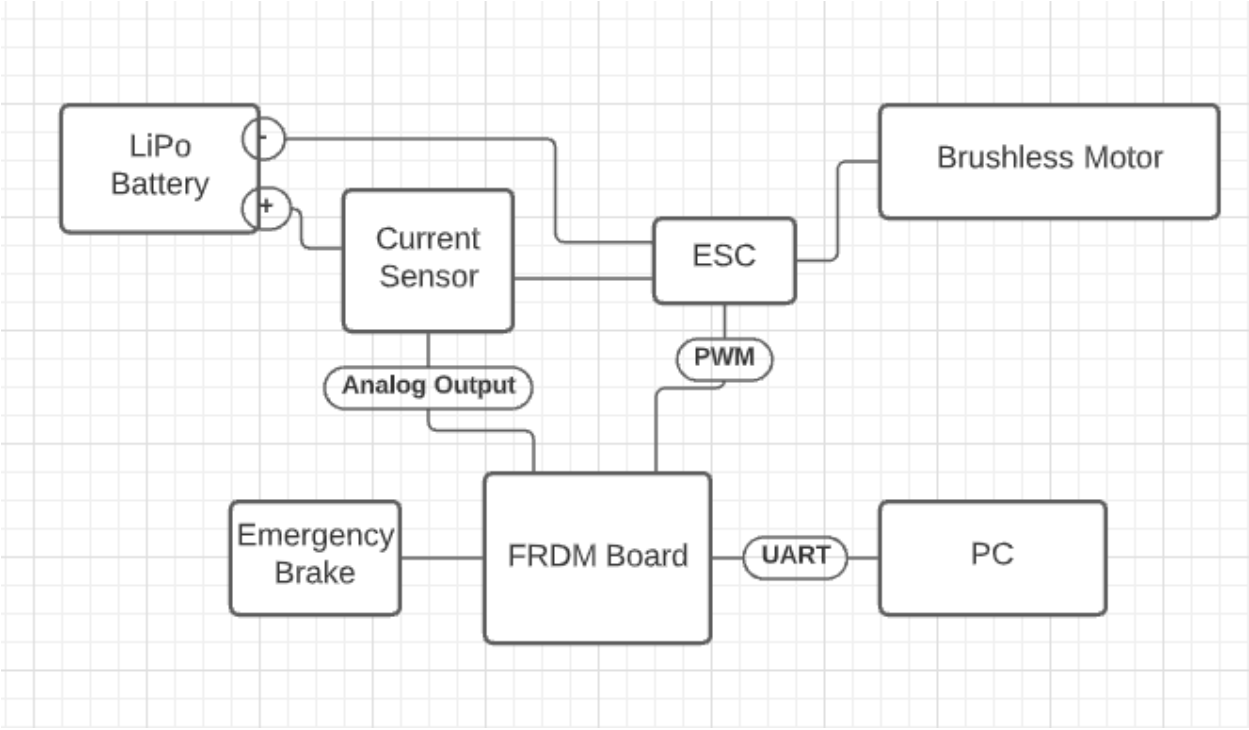
Microcontroller features used: Analog to digital conversion, UART, PWM generation, GPIO

Video Link

https://youtu.be/CZN7ZWjhhg4

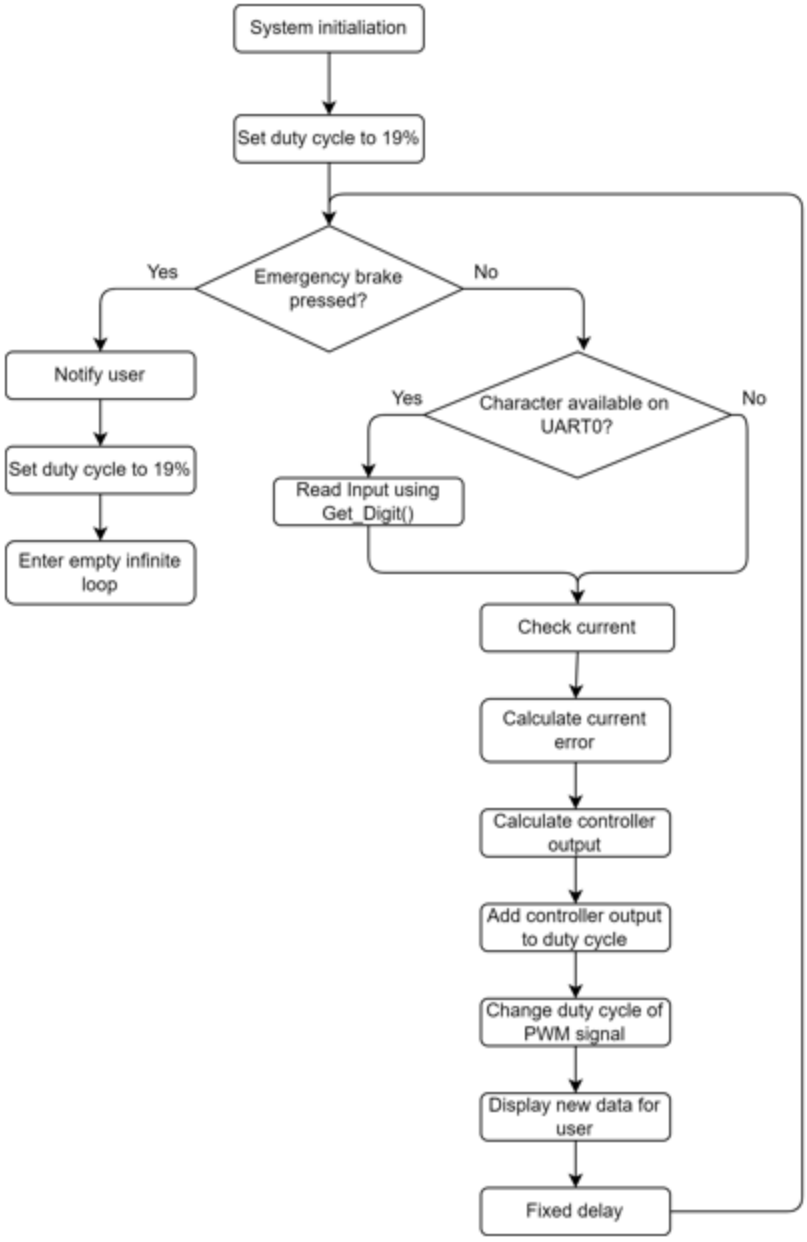
System Design

Block Diagram

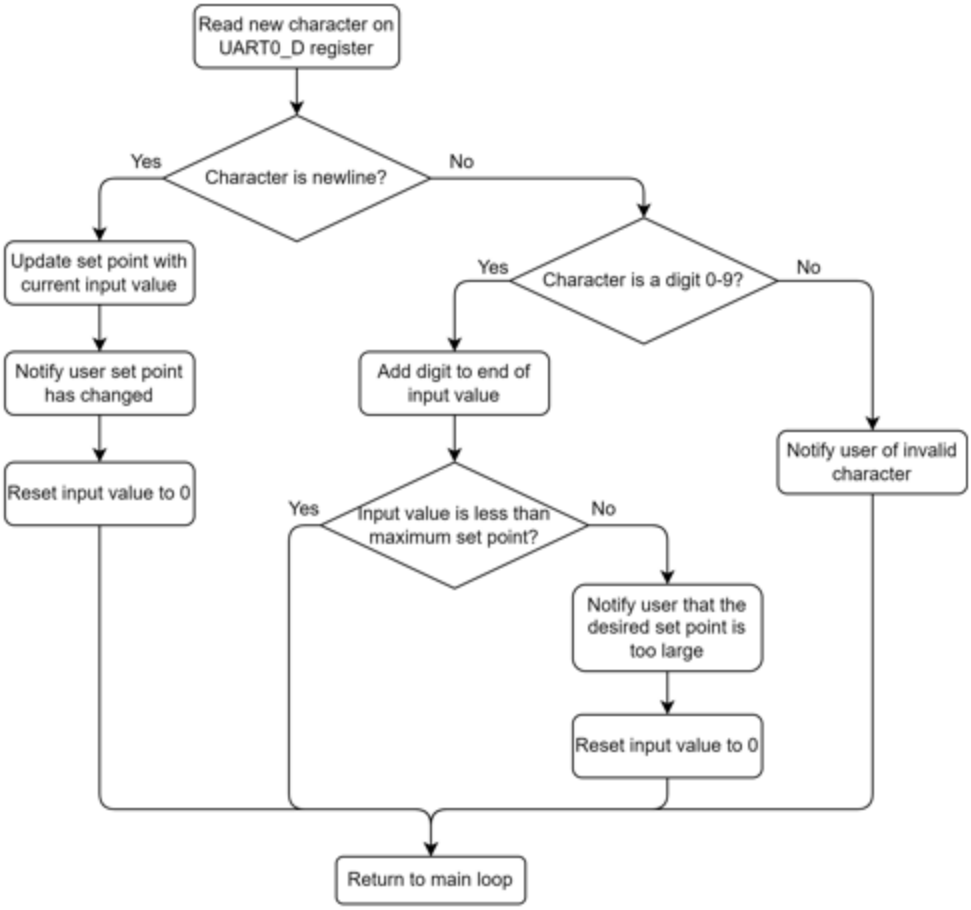


Program Flowcharts (See Software Design Section)

Flowchart 1 (Main Program Flowchart)



Flowchart 2 (User Input Handling Flowchart)

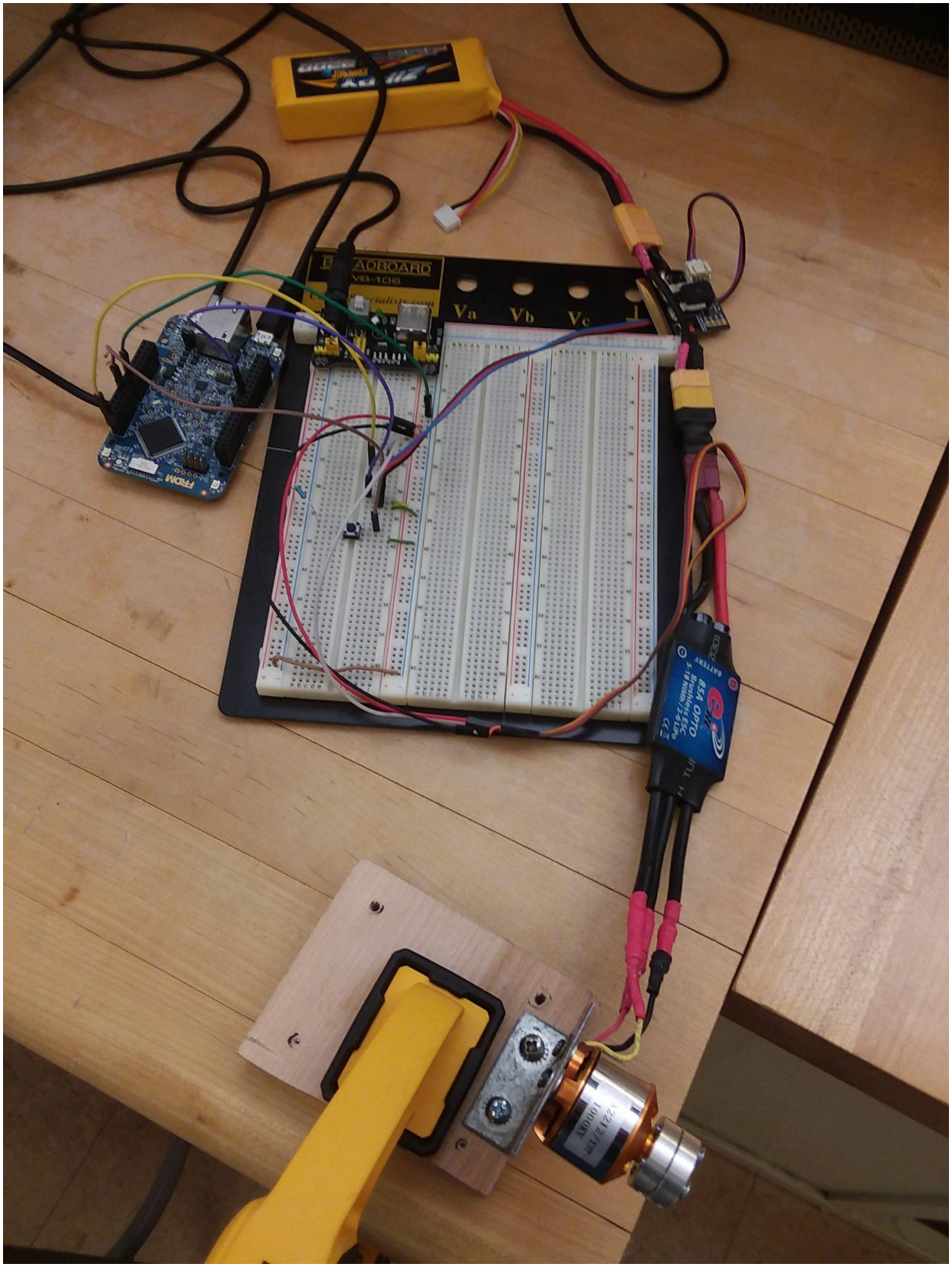


Implementation Details

Main hardware used:

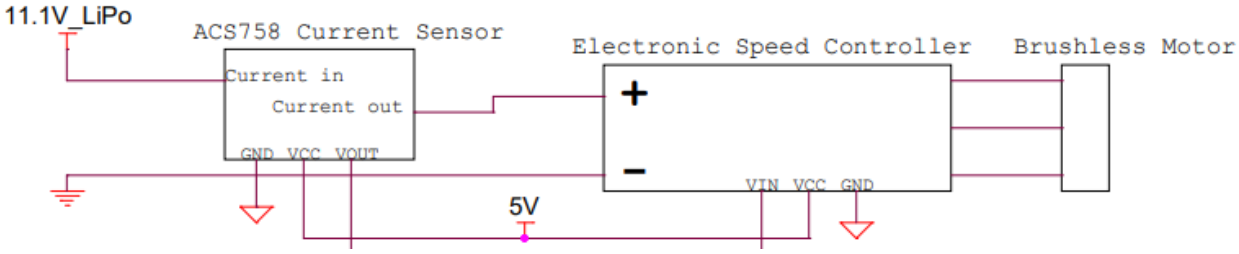
- Analog 50A Current Sensor
- 3S (11.1V) LiPo battery
- a2212/13t 1000kv brushless motor
- ERCE085P 85A Opto ESC
- FRDM K64F Evaluation Board
- Pushbutton switch
- Combined 5V and 3.3V power supply

Image of Final Design

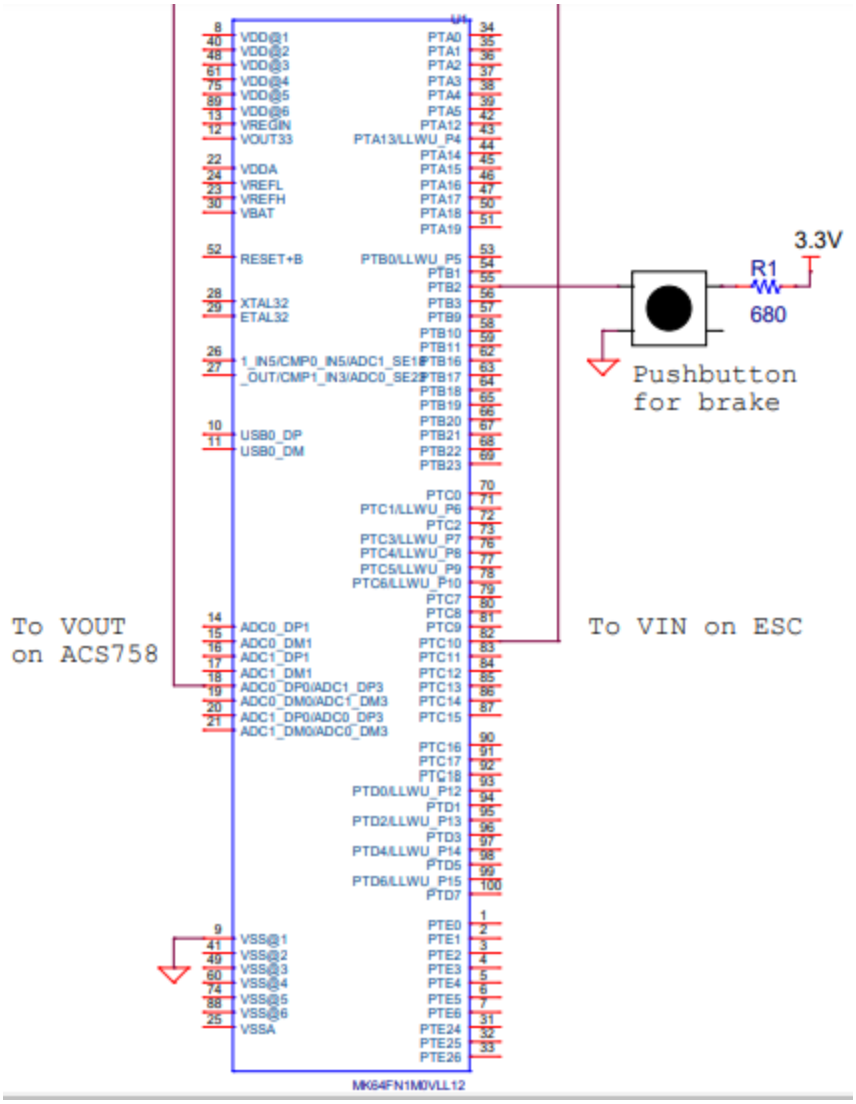

Va
Vb
Vc
60A OPTO
BRUSHLESS
5-12cell 7.4-50V

The physical connections made between the microcontroller and our motor driving circuit is shown in the image above and with the schematic for the microcontroller circuit shown below. Port B pin 2 on the k64f connects to our emergency brake button, port C pin 10 connects to the electronic speed controller (ESC) for the PWM signal generation, and port D pin 0 (ADC0_DP0) connects to the current sensor for analog to digital conversion, as the sensor outputs analog values. The digital grounds are also connected for the button, the current sensor and the ESC. For powering the motor, the electronic speed controller controls the voltage applied to the motor's three phases. The current going into the motor comes from the 11.1V LiPo battery, and passes through the current sensor which is in series with the rest of the motor driving circuit. This is shown in the schematic for the motor driving circuit below. The emergency brake works by pulling port B pin 2 high through a 680 ohm resistor when the button is not pressed, and directly connecting it to the ground when the button is pressed.

Schematic for the motor driving circuit



Schematic for the microcontroller circuit



Software Design

Flowchart 1 above gives the order in which the polling of inputs and the use of the PID controller to set a new duty cycle for the output PWM signal takes place. After system initialization, the PWM signal begins and is given a duty cycle of 19% in order to keep the motor off until a nonzero set point is specified by the user. An infinite loop is entered in which the system first checks whether the emergency brake button has been pressed, then checks if any input has been received through the UART connection, then samples the current sensor output and determines the new duty cycle, and finally displays updated information to the user.

To check whether the emergency brake button has been pressed, the value of the corresponding GPIO pin (pin 2 of port B) is queried as shown in the code snippet below. If the logic level at that pin is zero, then the brake triggers. This takes place in the Check_Emergency_Brake() function.

```
if ((GPIOB_PDIR & 0x00000004) == 0) {
```

```
        printf("Emergency brake is active\n");
        PWM_Set_Duty_Cycle(STARTUP_PWM_DC);
        while(1);
}
```

To check for input received through the UART connection, the UART0_RCFIFO register is polled. If the value of the register is greater than zero, then data is available. In this case, the Get_Digit() function, which operates based on flowchart 2 above, is executed. This involves a single character being read from the UART0 data register and processed. Since the user input is expected to be an integer value, when the character read is a digit 0-9, its contribution to the overall integer is determined using the following equation:

$$new\_input = (old\_input * 10) + new\_digit$$

If new_input is greater than the maximum set point that the system can allow (800 in this case), then the user is notified that they have exceeded the maximum set point and the total user input is reset (see code below). Otherwise, the function returns to the main loop until the next character is received and it is run again.

```
if (input > MAX_SETPOINT) {
        printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
        printf("User input exceeds maximum allowable set point\n");
        printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
        software_delay(4000000UL);
        input = 0;
        return;
}
```

When the character received is a newline character, it indicates that all digits have been entered and the set point can be updated (see code below). Any other non-digit character results in a notification that an invalid character has been entered. Whenever the user is given a notification regarding their input, a delay is introduced to give the user time to read it before new updates are printed to the console on the PC.

```
if (digit == 13) {
        // If newline is received, take the integer obtained from the user
        // and make it the new setpoint
        set_point = input;
        input = 0;
        printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
        printf("Set point changed: %d\n", set_point);
        printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
        software_delay(4000000UL);
}
```

Once the two sources of user input have been checked, the PID controller is executed within the PID_Loop() function. This function handles all steps from checking the current to changing the duty cycle of the PWM signal. To approximate the current flowing out of the battery, the current sensor is sampled 32 times and the average of those samples is taken. A lower limit of zero is given to the current reading to ensure that random noise does not result in a negative value. Next, the error is obtained by subtracting the current reading from the set point. As long as the set point is not set to zero, the error will then be used to calculate the proportional, derivative, and integral components of the controller output. The general form of the controller used is given by the following equation, where K_p, K_d, and K_i are constants, and err_i is the error recorded i iterations in the past:

$$output = K_p * err_0 + K_d * (err_0 - err_1) + K_i * \sum_{j=0}^{9} err_j$$

In this specific implementation, the derivative component is forced to be within the range [-0.1,0.1] to prevent excessive derivative kick during a set point change.

The output of the PID controller determines how much the duty cycle is changed from the duty cycle of the previous iteration. The exception to this is when the duty cycle has reached its upper or lower limit. The duty cycle is expressed as the percentage of the period for which the PWM signal is high, and after this value is passed into the function PWM_Set_Duty_Cycle(), it is converted to the 16-bit integer which, when passed into the PWM1_SetRatio16() function generated by Processor Expert, will produce a PWM signal with approximately the same duty cycle. The duty cycle generated this way is usually accurate to within +/- 0.2%.

The final step of displaying new data to the user takes place within the Update_UI() function. The values displayed to the user are the current read from the sensor in terms of the raw ADC units and in terms of milliamps, the error displayed in both forms, and the duty cycle that is currently being output. This is done using the following code:

```
char* new_pwm_string [50];
gcvt(duty_cycle, 6, new_pwm_string);
printf("-----------------------------\n");
printf("Current feedback raw: %hu \n", curr_current);
printf("Current feedback in milliamps: %d \n", current_in_milliamps(curr_current));
printf("Error: %d \n", curr_error);
printf("Error in milliamps: %d \n", current_in_milliamps(curr_error));
printf("New duty cycle: %s \n", new_pwm_string);
printf("-----------------------------\n");
```

Testing/Evaluation

When choosing values for K_p, K_d, and K_i that would allow the current drawn by the motor to track the set point, tests were initially run with both K_d and K_i set to zero until the controller could respond with sufficient speed to changes in the set point. Next, the derivative component

was introduced until the oscillations in the controller output were reduced to an acceptable level. However, we found that noise in the output of the current sensor could not be completely removed through averaging and this continued to have an impact on the effectiveness of the derivative component in stabilizing the output. Finally, the integral component was introduced to help bring the current result closer to the set point on average. Although the integral component helped, we did not find the effect to be very noticeable due to the amount of noise.

Challenges

Our main challenges were related to the software design, mainly due to the restriction of not being able to access parts of the Kinetis SDK directly when using Processor Expert. Our inability to easily control the duty cycle of PWM signals generated using our own code ultimately resulted in us needing to use Processor Expert. However, as a result we were unable to use code from the SDK that assists with setting up interrupts for GPIO pins. Although we tried using Processor Expert's modules to achieve the same result, we ultimately resorted to using polling for checking when the brake has been pressed.

Future Improvements

We had initially hoped to be able to make the emergency brake operate based on interrupts, and that is still something we would consider first as a future improvement. With more time we could have allowed for the emergency brake to be disengaged manually by pressing the button a second time rather than requiring the program to be reset. We could further improve the system by using seven segment displays to show the current being drawn from the battery.

Roles and Responsibilities of Group Members

This project required both equal challenge in hardware and software construction. Therefore, we both shared certain aspects of the hardware setup and software development. The overall hardware design was made by Jeremy, although the selection of pins to use on the FRDM board was a joint effort. Sefayet proposed code for the analog to digital conversion, the emergency brake, and the user input, and these were adapted into the main program. Jeremy was responsible for the PWM generation, the PID controller, and the output to the user. We worked together to test the system.

Conclusion

In the project, despite many difficulties using Processor Expert, we successfully accomplished our goal as originally intended. In technical merits, using the FRDM board we controlled a motor using feedback. More specifically, we correctly demonstrated how the FRDM board receives the desired current draw from a computer that is interfaced over UART. We also created a user interface that allowed the user to monitor current measured and the PWM duty cycle. And we accomplished it utilizing features such as analog to digital conversion, UART, pulse width modulation (PWM) generations, and GPIO of the microcontroller.

Appendix

Full Source Code

```
/* MODULE main */


/* Including needed modules to compile this module/procedure */
#include "Cpu.h"
#include "Events.h"
#include "Pins1.h"
#include "CsIO1.h"
#include "IO1.h"
#include "PWM1.h"
#include "PwmLdd1.h"
#include "TU1.h"
/* Including shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
#include "PDD_Includes.h"
#include "Init_Config.h"
/* User includes (#include below this line is not maintained by Processor
Expert) */
/************* Defined Constants *************/
#define MILLIVOLTS_PER_BIT 0.05035
#define MILLIAMPS_PER_MILLIVOLT 25
#define MILLIAMPS_PER_BIT 1.25881
// Duty cycle to hold the motor off
#define STARTUP_PWM_DC 19
// Lowest duty cycle that gets the motor to spin
#define MOTOR_START_DC 30
// Limit on the duty cycle that the ESC can run the motor at
#define DC_UPPER_LIMIT 80
// Limit on what the user can choose as the set point
#define MAX_SETPOINT 800
/************* Global Variables *************/
float duty_cycle = MOTOR_START_DC;
unsigned short curr_current = 0;
int curr_error = 0;
int set_point = 0;
const float Kp = 0.001;
const float Kd = 0.002;
const float Ki = 0.0001;
int last_error = 0;
int error_memory [10] = {0,0,0,0,0,0,0,0,0,0};
```

```
void software_delay(unsigned long delay)
{
    while (delay > 0) delay--;
}
void PWM_Set_Duty_Cycle(float desired_duty_cycle_percent) {
      // The duty cycle must be inverted to get the time low, then
      // multiply 2^16 by the ratio to get the input for the function
      // generated by processor expert; note that the division by 100
      // has alreay been taken into account
      float inverted_duty_cycle = 100.0 - desired_duty_cycle_percent;
      PWM1_SetRatio16((uint16_t)(655.35 * inverted_duty_cycle));
}
void Motor_Start_Up() {
      // Send a 19% duty cycle signal on start-up
      PWM_Set_Duty_Cycle(STARTUP_PWM_DC);
      software_delay(9000000UL);
}
unsigned short ADC_raw_val(void)
{
      ADC0_SC1A = 0x00;
      while(ADC0_SC2 & ADC_SC2_ADACT_MASK); // Conversion in progress
      while(!(ADC0_SC1A & ADC_SC1_COCO_MASK));
      return ADC0_RA;
}
unsigned short ADC_avg_val(void) {
      char j;
      unsigned long sum = 0;
      for (j = 0; j < 32; ++j) {
            unsigned short curr = ADC_raw_val();
            sum += curr;
      }
      // Bit shifting to divide by 16
      sum = sum >> 5;
      // 49843 corresponds to approximately 0 amps
      if (sum > 49843UL) {
            return (unsigned short)(sum - 49843UL);
      }
      else {
            return 0;
      }
}
int current_in_milliamps(int current_bits) {
```

```
	return (int)(current_bits * MILLIAMPS_PER_BIT);
}
void Check_Emergency_Brake() {
	// Non-interrupt solution to emergency brake
	if ((GPIOB_PDIR & 0x00000004) == 0) {
		printf("Emergency brake is active\n");
		PWM_Set_Duty_Cycle(STARTUP_PWM_DC);
		while(1);
	}
}
void PID_Loop() {
	curr_current = ADC_avg_val();
	// Needs to be a signed value
	curr_error = set_point - curr_current;

	// If the set point is specified as zero, then the motor is forced to
stop.
	// otherwise, the motor will be kept running by maintaining a minimum
duty cycle of 30%
	if (set_point == 0) {
		PWM_Set_Duty_Cycle(STARTUP_PWM_DC);
		duty_cycle = STARTUP_PWM_DC;
		return;
	}

	float proportional = Kp * curr_error;
	float derivative = Kd * (curr_error - last_error);
	// Limit how much the derivative term can react to a rapid
	// change in the amount of error to avoid overcompensation
	if (derivative > 0.1) {
		derivative = 0.1;
	}
	if (derivative < -0.1) {
		derivative = -0.1;
	}
	last_error = curr_error;
	// Integral term is the sum of the last 10 errors
	char i;
	float integral = 0;
	for (i = 0; i < 9; ++i) {
		error_memory[i] = error_memory[i+1];
		integral += error_memory[i];
	}
```

```
	// Index 9 stores the most recent error
	error_memory[9] = curr_error;
	integral += curr_error;
	integral = Ki * integral;

	float PID_control = proportional + derivative + integral;
	// The control signal from the PID controller determines how much
	// to change the duty cycle
	duty_cycle = duty_cycle + PID_control;

	// Establish upper and lower limits
	if (duty_cycle > DC_UPPER_LIMIT) {
		duty_cycle = DC_UPPER_LIMIT;
	}
	else if (duty_cycle < MOTOR_START_DC) {
		duty_cycle = MOTOR_START_DC;
	}

	PWM_Set_Duty_Cycle(duty_cycle);
}
void Update_UI() {
	char* new_pwm_string [50];
	gcvt(duty_cycle, 6, new_pwm_string);
	printf("-----------------------------\n");
	printf("Current feedback raw: %hu \n", curr_current);
	printf("Current feedback in milliamps: %d \n",
current_in_milliamps(curr_current));
	printf("Error: %d \n", curr_error);
	printf("Error in milliamps: %d \n",
current_in_milliamps(curr_error));
	printf("New duty cycle: %s \n", new_pwm_string);
	printf("-----------------------------\n");
}
void Get_Digit() {
	static int input = 0;
	char digit = UART0_D;
	// Only positive integers are accepted, so ignore any input
	// that is not a valid digit or newline character
	if (digit == 13) {
		// If newline is received, take the integer obtained from the
user
		// and make it the new setpoint
		set_point = input;
```

```
            input = 0;
            printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
            printf("Set point changed: %d\n", set_point);
            printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
            software_delay(4000000UL);
      }
      else if (digit >= '0' && digit <= '9') {
            // If digit is received, add it to the current user input and
check that
            // the input does not exceed the maximum allowed set point
            input = (input * 10) + (digit - 48);
            if (input > MAX_SETPOINT) {
                  printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
                  printf("User input exceeds maximum allowable set
point\n");
                  printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
                  software_delay(4000000UL);
                  input = 0;
                  return;
            }
      }
      else {
            printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
            printf("Invalid character received\n");
            printf("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n");
            software_delay(4000000UL);
      }
}
/*lint -save  -e970 Disable MISRA rule (6.3) checking. */
int main(void)
/*lint -restore Enable MISRA rule (6.3) checking. */
{
  /* Write your local variable definition here */

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!!
***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.
***/

  /* Write your code here */
  // Setup required for ADC
  SIM_SCGC6 |= SIM_SCGC6_ADC0_MASK; // 0x8000000u; Enable ADC0 Clock
```

```
  ADC0_CFG1 = 0x0C; // 16bits ADC; Bus Clock
  ADC0_SC1A = 0x1F; // Disable the module, ADCH = 11111
  // Setup required for emergency break
  SIM_SCGC5 |= SIM_SCGC5_PORTB_MASK;
  PORTB_PCR2 = 0x100;
  GPIOB_PDDR &= 0xFFFFFFFB;
  // Flush UART transmit and receive buffers
  UART0_CFIFO = 0XC0;

  printf("\n-----------------------------\n");
  printf("*****************************\n");
  printf("-----------------------------\n");

  Motor_Start_Up();

      while(1) {
            Check_Emergency_Brake();
            if (UART0_RCFIFO > 0) {
                  Get_Digit();
            }
            PID_Loop();
            Update_UI();
            software_delay(1000000UL);
      }

  /*** Don't write any code pass this line, or it will be deleted during
code generation. ***/
  /*** RTOS startup code. Macro PEX_RTOS_START is defined by the RTOS
component. DON'T MODIFY THIS CODE!!! ***/
  #ifdef PEX_RTOS_START
    PEX_RTOS_START();                  /* Startup of the selected RTOS.
Macro is defined by the RTOS component. */
  #endif
  /*** End of RTOS startup code.  ***/
  /*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! ***/
  for(;;){}
  /*** Processor Expert end of main routine. DON'T WRITE CODE BELOW!!! ***/
} /*** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/

/* END main */
```