Chris Payne
Yuan Chen
ELE 302 Stage 1, Speed Control Report
Team: **BLACK**
3.7.12

Final Hardware Design

The hardware component of our design involved four main modules, the RS-232 Interface, the Pulse Width Modulator, the Hall Sensor & Corresponding Interface, and the Voltage Regulator.

Voltage Regulator: This device was connected to our 7.2 Volt battery and was responsible for delivering 5 volts to the active components of our car's circuitry. The components of the voltage regulator board, including LM7805 chip, were given to us so we did not have to make any design decisions in terms of capacitors/IC's/etc. with respect to the Voltage Regulator board. We did decide to create three different male KK ports that could be easily accessed later on in the construction of the board by devices that needed 5 volts.

RS-232 Interface: This board was responsible for regulating the voltage difference between the computer's RS-232 logic and the car's Transistor to Transistor (TTL) logic. Incoming signals from the computer are regulated to on a 5 volt scale the TTL logic can process while outgoing signals to the computer are regulated to be on a 9 volt scale that RS-232 operates on. The components of the interface are given to us so we not need to make any involved design decisions for this board. It is worth noting though that one of our larger bugs occurred on this board when the transmit connection into the MAX232 chip became loose within the KK connector. With a repaired KK connection, we have encountered no new issues with this board.

Pulse Width Modulator(PWM): This board controls the energy that is delivered to the motor using a MOSFET chip and thus the actual speed of the car. The first design decision we had to make for this module was what MOSFET chip we would use based upon the motor's, MOSFET's, and processor board's specifications. The voltage supplied to the gate of the MOSFET by the processor output would be a maximum of 3.3 volts. Therefore, by examining the MOSFET's Voltage vs Current curves we could check if the corresponding source to drain current would supply too much power to the MOSFET. We determined the following calculations:

MOSFET	Gate Voltage from Processor [V]	Source to Drain Current [A]	MOSFET Max Power [W]	Approx Power @ 3.3 Volts [W]
MTP75N03HDL	3.3	45	150	324
MTP52N06VL	*3.3*	*25*	*188*	*180*
MTP75N0SHD	3.3	~ 0	150	~ 0
MTP75N06HD	3.3	~ 0	150	~ 0

Clearly the *MTP52N06VL* MOSFET delivered optimal parameters and thus we chose to use it in our PWM design.

The next design choice we had to make was what resistor we would attach to the gate of the MOSFET. This resistor had the job of protecting the processor in the event that the MOSFET failed and a short between the processor and either the ground or back EMF of the motor was created. The resistance couldn't be too large though because we did not want to have the output voltage drop too much over the resistor before reaching the gate. An additional consideration was that if we had too large of a resistor value, the RC time constant would be large and thus the voltage over the gate would not change as quickly as we intended. The back EMF could be as large 50 volts, which over a 20kΩ resistor during MOSFET breakdown conditions would deliver only 2.5 mA back to the processor, well below the 10 mA the processor is rated to handle.

Hall Sensor & Corresponding Interface: This module was responsible for relaying the actual number of wheel rotations to the processor so that it could accurately gauge its speed using our algorithm. This module was almost entirely predefined by the data sheets associate with the Hall sensor and comparator chip (LM393) and thus we really didn't need to make any design decisions. In mounting our Hall sensor, we decided to add some additional heat shrink to the connection between the 30 gauge wire and the Hall sensor itself. This gave us a nice steady surface to glue the hall sensor to the wheel suspension. It should also be noted that we tested our Hall sensor interface extensively before attaching it to the car so as to make sure our wire connections were intact and the orientation of the sensor was correct.

Software Design

From a broad prospective, we have three primary external inputs: The Hall Sensor, the RS-232 input, and the Interrupt Request(IRQA) button on the processor board. Internally, we have two clocks, one serving as a master clock and an Emergency clock to ensure the car has not entered stall state. The actuator of our system is the PWM output, which directly controls the speed of our motor.

IRQA: When pressed, the IRQA ("kill switch"), toggles the state of the car. If the car is running, all processes are terminated including any PWM signal, stopping the car. The history of the car is cleared by resetting all parameters to make it seem as if the car had never run. If the car is already in an off state, processes are initialized which allow the car to reach its target speed in a short time.

Hall Sensor: When one of the magnets passes by the Hall sensor, a signal is sent to the processor indicating this event, we will refer to the transmission of this signal as a 'click'. On the receipt of a click, we update the magnet index and retrieve the time that has elapsed since the last update that index. This corresponds to the time it has taken for one full revolution of our wheel. Since we know the radius of the wheel, we calculate the present speed of the car. This allows us to update the current error of the system (error defined as the target speed minus the measured speed of the car). After updating the error history, we also calculate the error derivative which is also used later on by our control algorithm. With our updated error and error derivative

information, the controller outputs a new PWM duty cycle in order to bring the car closer to its target speed.

RS-232: This communication channel allows us to vary the controller parameters of the car quickly, allowing us to tune the performance of the car efficiently. Additionally, the processor can transmit the state of all controller parameters back to our computer terminal to ensure values are what we think they are. Lastly, if the car is running, we can use this terminal to remotely put the car in the off state – equivalent to pressing the IRQA button.

Master Clock: This clock sends an internal interrupt every time the clock rises. This interrupt is caught by the our SpeedTimer_OnInterrupt() method, which increments its total time value by one period of the master clock each time it is catches an interrupt. We chose to set the period of this clock to be 1ms because it is mathematically simple and gives us enough resolution to determine our speed very accurately. The master clock also keeps track of the integral of the error signal by calculating the individual error area created after each clock rise and added it to the total previous area. We calculate the error integral here rather than on the Hall sensor interrupt, because we know the time dimension will always be constant (1 ms) and thus we just multiply this constant by the error value at the clock rise to get the new additional error integral. We did not choose a lower clock period to improve a resolution because we did not want to get overflow when counting the number of clock rises.

Emergency Clock: This clock counts the time between subsequent Hall sensor 'clicks'. If the target speed is not zero, the clock triggers an interrupt if 100 ms elapse between 'clicks'. This is indicative of a stall, and thus a relatively large duty cycle of 0.3 is sent via the PWM signal.

Algorithm: We store two, five element arrays called timeDifferences[] and errorHistory[]. One keep tracks of the time since the last occurrence of the index's associated magnet click. For example, index one, always holds the amount of time that has passed since magnet one last passed the Hall sensor. The other array keeps of the error that was present the last time the corresponding magnet click was detected. For example, index one will always hold the error present the last time magnet one produced a 'click'. This allows us to store a history containing the five most recent error values.

Using the modulus operator, we continuously cycle through the indexes of theses arrays in a circular fashion so that they are constantly updated on their corresponding magnet click. For example, on magnet click 5, we would update index 4 in both arrays; then on magnet click 6, we would update index 0 in both array, then on magnet click 7 we would update index 1 in both arrays, etc. After accessing each index, we reset the value of the index to zero (to prevent overflow and allow independent measurements between wheel revolutions). We use the error history array to calculate a derivate of the error averaged over the revolution. This is implementing by taking the present error value and subtracting it by the oldest stored error value. We divide this result by the time of a revolution, which is just stored in the current index of the timeDifferences[] array.

Controller: The controller is a PID controller that uses the above inputs we have discussed to output a PWM signal that will most likely cause the car to reach its target speed bases upon the error, derivative of the error, and integral of the error. We also take the open loop reference value

and add it as a constant to our PID calculations. This open loop offset was determined by testing various duty cycles and observing what speed they directly resulted in.

Determining the K Values: We slowly incremented our K_p value until our car reached an oscillatory state. Using the Ziegler Nichols method, we chose a K_p of 0.6 times this oscillatory K_p value. This resulted in an initial $K_p = 9.5$. We chose $K_d = 0.6$, which we guessed would prevent overshoot and oscillation. We also chose $K_i = 0$ initially, so that we could isolate the effects of raising the various constants. Using our RS-232 interface, we then slowly incremented these constant values until we reached the target speed on all three courses. The following table represents the values we tried during our testing phase:

K_p	**K_i**	**K_d**	**Speed Comments**	**Overall Comments**
9.5	10.4	1.2	N/A	Unstable
9.5	0.8	0.6	Too fast	Consistent
9.7	0.8	0.6	Too fast up/ too slow down ramp	Better
9.7	*1.2*	*0.6*	*Within requirements*	*Consistent and accurate*
9.9	1.2	0.6	Worse	Consistent but slightly off

From our testing, we settled on using **$K_p = 9.7$, $K_i = 1.2$,** and **$K_d = 0.6$**. These values gave us a perfect combination of a quick rise time, yet enough derivate control that our overshoot and settling time were minimized. Additionally, out integral control helped zero out our steady state.

Lastly, we noticed that our car was consistently running the courses too fast. This indicated that our speed control was working fine, but our target speed was not accurately calibrated due to inaccuracies in the measurement of the wheel radius. To remedy this, we lowered our target speed slightly from 3.0 to 2.895 feet/s which gave us both consistent and accurate speed!

Bugs/Limitations: At this time there are no foreseeable issues with our design or how we have implemented it.

Below, please find our implementation and interface files for Controller.c, ConversionMethods.c, SpeedControl.c, and Events.c

```
/***
 * Controller.c
 *
 * ELE 302
 * Yuan Chen, Chris Payne
 *
 * A closed loop system for controlling motor speed
 **/

 #include "Controller.h"
 #include "SpeedControl.h"
 #include "ConversionMethods.h"

 unsigned int getRatio(int useClosedLoop)
 {
        double currentError;
        double desiredSpeed;
        double sensorSpeed;
        double dutyCycle = 0.0;

        desiredSpeed = getRefSpeed();
        sensorSpeed = getMeasuredSpeed();

        //set a value of u_0;
        dutyCycle = (desiredSpeed)/24.437;

        currentError = desiredSpeed - sensorSpeed;

        /*
        errorDeriv = 0.0;

        if (getTimeDifference((getMagIndex()+3)%5) != 0.0)
        {
                        errorDeriv = 1000.0*(currentError -
getError())/getTimeDifference((getMagIndex()+3)%5);
        }*/

        setError(currentError);

        if (useClosedLoop == 0) //use only open loop control
        {
                if (dutyCycle > 1.0)
```

```
		{
			dutyCycle = 1.0;
		}
		if (dutyCycle < 0.0)
		{
			dutyCycle = .05;
		}

		return dutyCycleToRatio(dutyCycle);
	}

	dutyCycle = dutyCycle + (currentError * getKp()/24.437) + (getErrorDeriv() *
getKd()/24.437) + (getErrorIntegral() * getKi()/24.437);

	return dutyCycleToRatio(dutyCycle);
 }
```

****

```
/***
 * Controller.h
 *
 * ELE 302
 * Yuan Chen, Chris Payne
 *
 * A closed loop system for controlling motor speed
 **/

 #ifndef _CNTRL
 #define _CNTRL

 // Returns a PWM duty cycle ratio as an unsigned integer
 // using the feedback between the desiredSpeed and measuredSpeed
 unsigned int getRatio(int useClosedLoop);

 #endif
```

****

```
// ConversionMethods.c
//
// ELE 302
```

```
// Yuan Chen, Christopher Payne
//
// Methods for converting between physical units and
// input parameters for the microprocessor

#include "ConversionMethods.h"

// Returns as an integer the number of sensor clicks needed
// to travel distance feet
int distanceToClicks(double distance)
{
	double numClicks = distance*(12.0)*(2.54)/(3.896);
	return (int)numClicks;
}

// Returns as an unsigned integer (from 0x0000 to 0xFFFF)
// the ratio parameter required to achieve a duty cycle
// of percentage cycle
//
// cycle is from 0 to 1 inclusive
unsigned int dutyCycleToRatio(double cycle)
{
	double ratio = (1.0 - cycle) * 65535.0;
	if ((cycle < 0.0) || (cycle > 1.0))
	{
		return (0xFFFF);
	}
	return (unsigned int)ratio;
}
```

****

```
// ConversionMethods.h
//
// ELE 302
// Yuan Chen, Christopher Payne
//
// Methods for converting between physical units and
// input parameters for the microprocessor

#ifndef _CNVR_MTHD
#define _CNVR_MTHD

// Returns as an integer the number of sensor clicks needed
```

```
// to travel distance feet
int distanceToClicks(double distance);

// Returns as an unsigned integer (from 0x0000 to 0xFFFF)
// the ratio parameter required to achieve a duty cycle
// of percentage cycle
//
// cycle is from 0 to 1 inclusive
unsigned int dutyCycleToRatio(double cycle);

#endif



**************************************************************************
/** ###################################################################
**     Filename  : Events.C
**     Project   : SpeedControl
**     Processor : 56F801FA60
**     Component : Events
**     Version   : Driver 01.03
**     Compiler  : Metrowerks DSP C Compiler
**     Date/Time : 3/2/2012, 6:44 PM
**     Abstract  :
**         This is user's event module.
**         Put your event handler code here.
**     Settings  :
**     Contents  :
**         No public methods
**
** ###################################################################*/
/* MODULE Events */

#include "Cpu.h"
#include "Events.h"
#include "SpeedControl.h"
#include "Controller.h"
#include "ConversionMethods.h"

/* User includes (#include below this line is not maintained by Processor Expert) */

/*
**
============================================================
**     Event      :  SpeedTimer_OnInterrupt (module Events)
**
```

```
**     Component   :  SpeedTimer [TimerInt]
**     Description :
**         When a timer interrupt occurs this event is called (only
**         when the component is enabled - <Enable> and the events are
**         enabled - <EnableEvent>). This event is enabled only if a
**         <interrupt service/event> is enabled.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve registers'
property */
                        /* is set to 'yes' (#pragma interrupt saveall is generated before the ISR)      */
void SpeedTimer_OnInterrupt(void)
{
          int i;
          //int useClosedLoop;
          for (i = 0; i < 5; i++)
          {
                    setTimeDifference(i, getTimeDifference(i)+1);
          }

          setErrorIntegral(getErrorIntegral() + (getRefSpeed() - getMeasuredSpeed())/1000.0);

}

/*
**
===================================================================
**     Event       :  HallSensorInterrupt_OnInterrupt (module Events)
**
**     Component   :  HallSensorInterrupt [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve registers'
property */
                        /* is set to 'yes' (#pragma interrupt saveall is generated before the ISR)      */
void HallSensorInterrupt_OnInterrupt(void)
{
```

```
  int timeTicks; //time difference in milliseconds
  double speed;
  int useClosedLoop;
  double currentError;

  setNoClickTime(0);
  setMagIndex((getMagIndex() % 5) + 1);
  setTotalClicks(getTotalClicks()+1);
  timeTicks = getTimeDifference(getMagIndex()-1);

  if (timeTicks == 0)
  {
        setMeasuredSpeed(0.0);
        return;
  }

  speed = (19.47/(double)timeTicks) * 1000.0/(2.54 * 12.0);

  setMeasuredSpeed(speed);
  currentError = getRefSpeed() - getMeasuredSpeed();

  setErrorDeriv((currentError - getErrorHistory(getMagIndex()-1))*1000.0/(double)timeTicks);
  setErrorHistory(currentError, getMagIndex()-1);
  useClosedLoop = (getTotalClicks() > 10) ? 1 : 0;
  MotorPWM_SetRatio16(getRatio(useClosedLoop));


  setTimeDifference(getMagIndex()- 1, 0);
  AS1_ClearTxBuf();
  //AS1_SendChar((unsigned int)(48+(int)speed));
}


/*
**
===================================================================
**     Event      :  KillSwitch_OnInterrupt (module Events)
**
**     Component   :  KillSwitch [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
```

```
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve registers'
property */
                        /* is set to 'yes' (#pragma interrupt saveall is generated before the ISR)      */
void KillSwitch_OnInterrupt(void)
{
        if (getOpStatus() == MOTOR_ON)
        {
                int i;
                setOpStatus(MOTOR_OFF);
                HallSensorInterrupt_Disable();
                SpeedTimer_Disable();
                EmergencyNoStopTimer_Disable();
                MotorPWM_SetRatio16(0xFFFF);

                for (i = 0; i < 5; i++)
                {
                        setTimeDifference(i, 0);
                }
                setMeasuredSpeed(0.0);
                setError(0.0);
                setErrorIntegral(0.0);
                setMagIndex(0);
                setNoClickTime(0);
                setTotalClicks(0);
        }
        else
        {
                setOpStatus(MOTOR_ON);
                HallSensorInterrupt_Enable();
                SpeedTimer_Enable();
                EmergencyNoStopTimer_Enable();
                MotorPWM_SetRatio16(getRatio(0));

        }

}


/*
**
===================================================================
**     Event      :  AS1_OnError (module Events)
**
**     Component  :  AS1 [AsynchroSerial]
**     Description :
```

```
**         This event is called when a channel error (not the error
**         returned by a given method) occurs. The errors can be
**         read using <GetError> method.
**         The event is available only when the <Interrupt
**         service/event> property is enabled.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve registers'
property */
                    /* is set to 'yes' (#pragma interrupt saveall is generated before the ISR)      */
void AS1_OnError(void)
{
  /* Write your code here ... */
}

/*
**
============================================================================
**     Event       :  AS1_OnRxChar (module Events)
**
**     Component   :  AS1 [AsynchroSerial]
**     Description :
**         This event is called after a correct character is
**         received.
**         The event is available only when the <Interrupt
**         service/event> property is enabled and either the
**         <Receiver> property is enabled or the <SCI output mode>
**         property (if supported) is set to Single-wire mode.
**         Version specific information for Freescale 56800
**         derivatives ]
**         DMA mode:
**         If DMA controller is available on the selected CPU and
**         the receiver is configured to use DMA controller then
**         this event is disabled. Only OnFullRxBuf method can be
**         used in DMA mode.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve registers'
property */
                    /* is set to 'yes' (#pragma interrupt saveall is generated before the ISR)      */
```

```
void AS1_OnRxChar(void)
{

	int i;
	AS1_TComData *a;
	unsigned int out;

	setOpStatus(MOTOR_OFF);
	HallSensorInterrupt_Disable();
	SpeedTimer_Disable();
	EmergencyNoStopTimer_Disable();
	MotorPWM_SetRatio16(0xFFFF);

	for (i = 0; i < 5; i++)
	{
		setTimeDifference(i, 0);
	}

	setMeasuredSpeed(0.0);
	setError(0.0);
	setErrorIntegral(0.0);
	setMagIndex(0);
	setNoClickTime(0);
	setTotalClicks(0);
	//The  following sequence is how we incremented/decremented our K values using RS-
232
	AS1_ClearTxBuf();

	AS1_RecvChar(a);

	if (a[0] == 'a')
	{
		setKp(getKp() + 0.2);
		out = (getKp() / 0.2);
		AS1_SendChar(' ');
		AS1_SendChar('p');
		AS1_SendChar('=');
		AS1_SendChar(out + 33);
	}

	if (a[0] == 'z')
	{
		setKp(getKp() - 0.2);
		out = (getKp() / 0.2);
		AS1_SendChar(' ');
		AS1_SendChar('p');
```

```
        AS1_SendChar('=');
        AS1_SendChar(out + 33);
}

if (a[0] == 's')
{
        setKd(getKd() + 0.2);
        out = (getKd() / 0.2);
        AS1_SendChar(' ');
        AS1_SendChar('d');
        AS1_SendChar('=');
        AS1_SendChar(out + 33);
}

if (a[0] == 'x')
{
        setKd(getKd() - 0.2);
        out = (getKd() / 0.2);
        AS1_SendChar(' ');
        AS1_SendChar('d');
        AS1_SendChar('=');
        AS1_SendChar(out + 33);
}

if (a[0] == 'd')
{
        setKi(getKi() + 0.2);
        out = (getKi() / 0.2);
        AS1_SendChar(' ');
        AS1_SendChar('i');
        AS1_SendChar('=');
        AS1_SendChar(out + 33);
}

if (a[0] == 'c')
{
        setKi(getKi() - 0.2);
        out = (getKi() / 0.2);
        AS1_SendChar(' ');
        AS1_SendChar('i');
        AS1_SendChar('=');
        AS1_SendChar(out + 33);
}

if (a[0] == 'p')
{
```

```
            out = (getKp() / 0.2);
            AS1_SendChar(' ');
            AS1_SendChar('p');
            AS1_SendChar('~');
            AS1_SendChar(out + 33);
        }

        if (a[0] == 'i')
        {
            out = (getKi() / 0.2);
            AS1_SendChar(' ');
            AS1_SendChar('i');
            AS1_SendChar('~');
            AS1_SendChar(out + 33);
        }

        if (a[0] == 'f')
        {
            out = (getKd() / 0.2);
            AS1_SendChar(' ');
            AS1_SendChar('d');
            AS1_SendChar('~');
            AS1_SendChar(out + 33);
        }


        AS1_ClearRxBuf();

}

/*
**
===================================================================
**     Event      :  AS1_OnTxChar (module Events)
**
**     Component  :  AS1 [AsynchroSerial]
**     Description :
**         This event is called after a character is transmitted.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve registers'
property */
                    /* is set to 'yes' (#pragma interrupt saveall is generated before the ISR)      */
```

```
void AS1_OnTxChar(void)
{
  // Use this to adjust various k values
}

/*
**
============================================================
**     Event      :  AS1_OnFullRxBuf (module Events)
**
**     Component  :  AS1 [AsynchroSerial]
**     Description :
**         This event is called when the input buffer is full;
**         i.e. after reception of the last character
**         that was successfully placed into input buffer.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve registers'
property */
                         /* is set to 'yes' (#pragma interrupt saveall is generated before the ISR)      */
void AS1_OnFullRxBuf(void)
{
  /* Write your code here ... */
}

/*
**
============================================================
**     Event      :  AS1_OnFreeTxBuf (module Events)
**
**     Component  :  AS1 [AsynchroSerial]
**     Description :
**         This event is called after the last character in output
**         buffer is transmitted.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve registers'
property */
                         /* is set to 'yes' (#pragma interrupt saveall is generated before the ISR)      */
void AS1_OnFreeTxBuf(void)
```

```
{
  /* Write your code here ... */
}

/*
**
============================================================
**     Event      :  EmergencyNoStopTimer_OnInterrupt (module Events)
**
**     Component   :  EmergencyNoStopTimer [TimerInt]
**     Description :
**         When a timer interrupt occurs this event is called (only
**         when the component is enabled - <Enable> and the events are
**         enabled - <EnableEvent>). This event is enabled only if a
**         <interrupt service/event> is enabled.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve registers'
property */
                         /* is set to 'yes' (#pragma interrupt saveall is generated before the ISR)      */
void EmergencyNoStopTimer_OnInterrupt(void)
{
  setNoClickTime(getNoClickTime() + 1);

  if (getNoClickTime() > 100)
  {
          MotorPWM_SetRatio16(dutyCycleToRatio(.3));
  }
}

/* END Events */

/*
** ###################################################################
**
**     This file was created by Processor Expert 3.00 [04.35]
**     for the Freescale 56800 series of microcontrollers.
**
** ###################################################################
*/


*************************************************************************************
```

```
/** ###################################################################
**     Filename  : Events.H
**     Project   : SpeedControl
**     Processor : 56F801FA60
**     Component : Events
**     Version   : Driver 01.03
**     Compiler  : Metrowerks DSP C Compiler
**     Date/Time : 3/2/2012, 6:44 PM
**     Abstract  :
**         This is user's event module.
**         Put your event handler code here.
**     Settings  :
**     Contents  :
**         No public methods
**
** ###################################################################*/

#ifndef __Events_H
#define __Events_H
/* MODULE Events */

#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
#include "PE_Timer.h"
#include "KillSwitch.h"
#include "HallSensorInterrupt.h"
#include "SpeedTimer.h"
#include "AS1.h"
#include "MotorPWM.h"
#include "EmergencyNoStopTimer.h"


void SpeedTimer_OnInterrupt(void);
/*
**
** ===================================================================
**     Event       :  SpeedTimer_OnInterrupt (module Events)
**
**     Component   :  SpeedTimer [TimerInt]
**     Description :
**         When a timer interrupt occurs this event is called (only
**         when the component is enabled - <Enable> and the events are
**         enabled - <EnableEvent>). This event is enabled only if a
**         <interrupt service/event> is enabled.
```

```
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/

void HallSensorInterrupt_OnInterrupt(void);
/*
**
============================================================
**     Event       :  HallSensorInterrupt_OnInterrupt (module Events)
**
**     Component   :  HallSensorInterrupt [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/
void KillSwitch_OnInterrupt(void);
/*
**
============================================================
**     Event       :  KillSwitch_OnInterrupt (module Events)
**
**     Component   :  KillSwitch [ExtInt]
**     Description :
**         This event is called when an active signal edge/level has
**         occurred.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/
void AS1_OnError(void);
/*
**
============================================================
**     Event       :  AS1_OnError (module Events)
**
**     Component   :  AS1 [AsynchroSerial]
**     Description :
**         This event is called when a channel error (not the error
**         returned by a given method) occurs. The errors can be
```

```
**         read using <GetError> method.
**         The event is available only when the <Interrupt
**         service/event> property is enabled.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/

void AS1_OnRxChar(void);
/*
**
============================================================
**     Event       :  AS1_OnRxChar (module Events)
**
**     Component   :  AS1 [AsynchroSerial]
**     Description :
**         This event is called after a correct character is
**         received.
**         The event is available only when the <Interrupt
**         service/event> property is enabled and either the
**         <Receiver> property is enabled or the <SCI output mode>
**         property (if supported) is set to Single-wire mode.
**         Version specific information for Freescale 56800
**         derivatives ]
**         DMA mode:
**         If DMA controller is available on the selected CPU and
**         the receiver is configured to use DMA controller then
**         this event is disabled. Only OnFullRxBuf method can be
**         used in DMA mode.
**     Parameters  : None
**     Returns     : Nothing
**
============================================================
*/

void AS1_OnTxChar(void);
/*
**
============================================================
**     Event       :  AS1_OnTxChar (module Events)
**
**     Component   :  AS1 [AsynchroSerial]
**     Description :
**         This event is called after a character is transmitted.
**     Parameters  : None
```

```
**     Returns     : Nothing
**
===============================================================
*/

void AS1_OnFullRxBuf(void);
/*
**
===============================================================
**     Event       :  AS1_OnFullRxBuf (module Events)
**
**     Component   :  AS1 [AsynchroSerial]
**     Description :
**         This event is called when the input buffer is full;
**         i.e. after reception of the last character
**         that was successfully placed into input buffer.
**     Parameters  : None
**     Returns     : Nothing
**
===============================================================
*/

void AS1_OnFreeTxBuf(void);
/*
**
===============================================================
**     Event       :  AS1_OnFreeTxBuf (module Events)
**
**     Component   :  AS1 [AsynchroSerial]
**     Description :
**         This event is called after the last character in output
**         buffer is transmitted.
**     Parameters  : None
**     Returns     : Nothing
**
===============================================================
*/

void EmergencyNoStopTimer_OnInterrupt(void);
/*
**
===============================================================
**     Event       :  EmergencyNoStopTimer_OnInterrupt (module Events)
**
**     Component   :  EmergencyNoStopTimer [TimerInt]
**     Description :
```

```
**         When a timer interrupt occurs this event is called (only
**         when the component is enabled - <Enable> and the events are
**         enabled - <EnableEvent>). This event is enabled only if a
**         <interrupt service/event> is enabled.
**     Parameters  : None
**     Returns     : Nothing
**
===================================================================
*/

/* END Events */
#endif /* __Events_H*/

/*
** ###################################################################
**
**     This file was created by Processor Expert 3.00 [04.35]
**     for the Freescale 56800 series of microcontrollers.
**
** ###################################################################
*/


*********************************************************************************
/** ###################################################################
**     Filename  : SpeedControl.C
**     Project   : SpeedControl
**     Processor : 56F801FA60
**     Version   : Driver 01.14
**     Compiler  : Metrowerks DSP C Compiler
**     Date/Time : 3/2/2012, 6:44 PM
**     Abstract  :
**         Main module.
**         This module contains user's application code.
**     Settings  :
**     Contents  :
**         No public methods
**
** ###################################################################*/
/* MODULE SpeedControl */


/* Including needed modules to compile this module/procedure */
#include "Cpu.h"
#include "Events.h"
#include "KillSwitch.h"
```

```
#include "HallSensorInterrupt.h"
#include "SpeedTimer.h"
#include "AS1.h"
#include "MotorPWM.h"
#include "EmergencyNoStopTimer.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 "SpeedControl.h"

int opStatus = MOTOR_ON;

/* Keeps track of the time since the magnet was last detected*/
int timeDifferences[] = {0, 0, 0, 0, 0};

/* Tracks the current magnet pulse*/
int magIndex = 0;
int noClickTime = 0;

/* Keeps track of the measured speed in feet per second */
double measuredSpeed = 0.0;
double error = 0.0;
double errorIntegral = 0.0;

double errorHistory[] = {0.0, 0.0, 0.0, 0.0, 0.0};
double errorDerivative = 0.0;

double kp = 9.7;
double kd = 0.6;
double ki = 1.2;

int totalClicks = 0;

void main(void)
{
  /* 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 */

  for(;;) {}
```

```
}

/* return the derivative of the error */
double getErrorDeriv()
{
	return errorDerivative;
}

/* set the derivative of the error */
void setErrorDeriv(double errorDeriv)
{
	errorDerivative = errorDeriv;
}

/* return the history of the error */
double getErrorHistory(int index)
{
	return errorHistory[index];
}

/* set the history of the error */
void setErrorHistory(double errorHist, int index)
{
	errorHistory[index] = errorHist;
}


/* return the operation status of the car */
int getOpStatus()
{
	return opStatus;
}

/* set the operation status of the car */
void setOpStatus(int status)
{
	opStatus = status;
}

/* return the integral of the error */
double getErrorIntegral()
{
	return errorIntegral;
}

/* set the integral of the error */
```

```
void setErrorIntegral(double ei)
{
	errorIntegral = ei;
}

/* return the time since the last click */
int getNoClickTime()
{
	return noClickTime;
}

/* set the time since the last click */
void setNoClickTime(int n)
{
	noClickTime = n;
}


/* Return the total clicks traveled */
int getTotalClicks()
{
	return totalClicks;
}

/* Set the total clicks traveled */
void setTotalClicks(int c)
{
	totalClicks = c;
}

/* Return the reference speed */
double getRefSpeed()
{
	return 2.895;
}

/* Returns the current magIndex*/
int getMagIndex()
{
	return magIndex;
}

/* Set the magIndex to i*/
void setMagIndex(int i)
{
	magIndex = i;
```

```
}

/* Return the time difference in the timeDifference array
   at index */
int getTimeDifference(int index)
{
        return timeDifferences[index];
}

/* Set the timeDifference index to value*/
void setTimeDifference(int index, int value)
{
        timeDifferences[index] = value;
}

/* Return the measured speed in feet/sec */
double getMeasuredSpeed()
{
        return measuredSpeed;
}

/* Set the measured speed in feet/sec */
void setMeasuredSpeed(double s)
{
        measuredSpeed = s;
}

/* Set Kp */
void setKp(double k)
{
        kp = k;
}

/* Set Kd */
void setKd(double k)
{
        kd = k;
}

/* Set Ki */
void setKi(double k)
{
        ki = k;
}

/* Get kp */
```

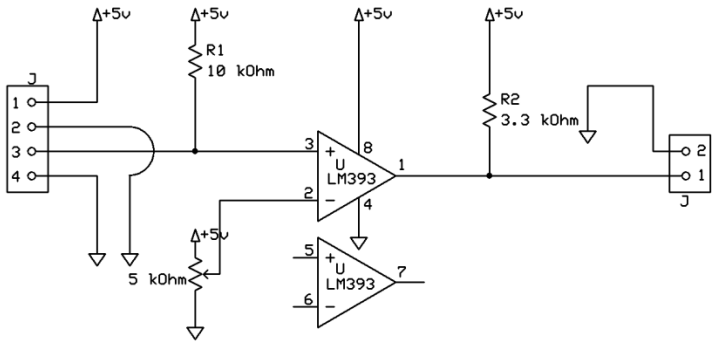
```
double getKp()
{
	return kp;
}

/* Get Kd */
double getKd()
{
	return kd;
}

/* Get Ki */
double getKi()
{
	return ki;
}

/* Return the error in feet/sec^2 */
double getError()
{
	return error;
}

/* Set the error in feet/sec^2 */
void setError(double ed)
{
	error = ed;
}

/* END ClosedLoopSpeedControl */
/*
** ###################################################################
**
**     This file was created by Processor Expert 3.00 [04.35]
**     for the Freescale 56800 series of microcontrollers.
**
** ###################################################################
*/


*****************************************************************************
*
/***
* SpeedControl.h
*
* ELE 302
```

```
* Yuan Chen, Chris Payne
*
* The interface file for the closed loop speed control methods
**/

#ifndef _SPDCTRL
#define _SPDCTRL

enum
{
	MOTOR_ON = 0,
	MOTOR_OFF = 1
};

/* return the derivative of the error */
double getErrorDeriv();

/* set the derivative of the error */
void setErrorDeriv(double errorDeriv);

/* return the history of the error */
double getErrorHistory(int index);

/* set the history of the error */
void setErrorHistory(double errorHist, int index);

/* return the operation status of the car */
int getOpStatus();

/* set the operation status of the car */
void setOpStatus(int status);

/* return the integral of the error */
double getErrorIntegral();

/* set the integral of the error */
void setErrorIntegral(double ei);

/* return the time since the last click */
int getNoClickTime();

/* set the time since the last click */
void setNoClickTime(int n);

/* Return the total clicks traveled */
int getTotalClicks();
```

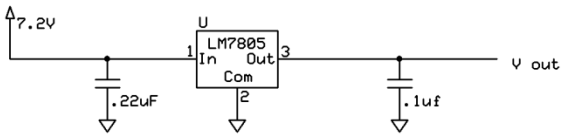
```
/* Set the total clicks traveled */
void setTotalClicks(int c);

/* Return the desired reference speed */
double getRefSpeed();

/* Returns the current magIndex*/
int getMagIndex();

/* Set the magIndex to i*/
void setMagIndex(int i);

/* Return the time difference in the timeDifference array
at index */
int getTimeDifference(int index);

/* Set the timeDifference index to value*/
void setTimeDifference(int index, int value);

/* Return the measured speed in feet/sec */
double getMeasuredSpeed();

/* Set the measured speed in feet/sec */
void setMeasuredSpeed(double s);

/* Return the error in feet/sec^2 */
double getError();

/* Set the error in feet/sec^2 */
void setError(double ed);

/* Set Kp */
void setKp(double k);

/* Set Kd */
void setKd(double k);

/* Set Ki */
void setKi(double k);

/* Get kp */
double getKp();

/* Get Kd */
double getKd();
```

```
/* Get Ki */
double getKi();

#endif
```

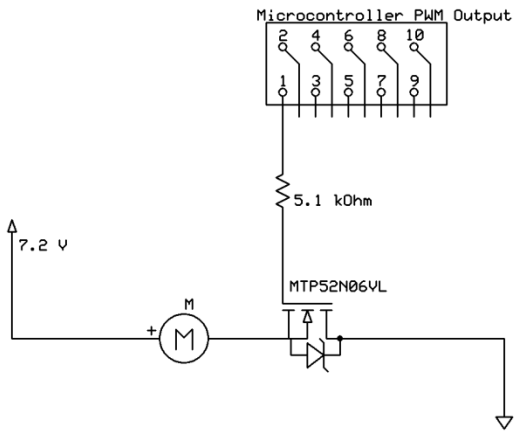
****



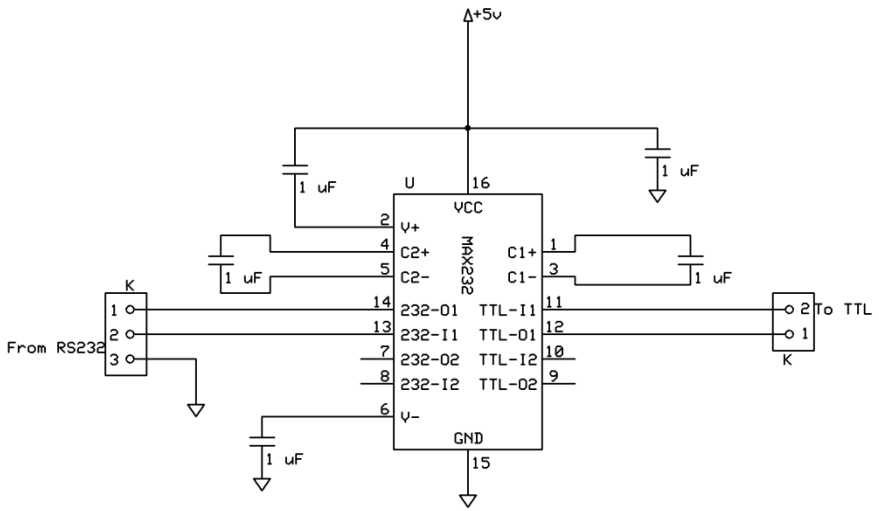
Team Black		
Hall Effect Interface Board		
Yuan Chen, Chris Payne	Rev 1.0 3/7/2012	



Team Black		
Power Regulator Circuit		
Yuan Chen, Chris Payne	Rev 1.0 3/7/2012	



Team Black		
PWM Motor Drive Circuit		
Yuan Chen, Chris Payne	Rev 1.0 3/7/2012	



Team Black		
RS232 Interface		
Yuan Chen, Chris Payne	Rev 1.0 3/7/2012	