Smart Motor Library

Version 1.00

Prepared by: James Pearman

Date: 11 March 2013

1st Edition Rev 2.1

Introduction

The Smart Motor library is a series of C code functions and data structures that provide additional functionality to ROBOTC for the VEX cortex. The library is distributed as a single header file that is included at the top of the main ROBOTC source file by means of an #include statement. The library provides the following functionality.

- The automatic calculation of actual motor speed based on encoder value changes.
- The automatic calculation of motor current based on measured motor speed and motor control value.
- The estimation of PTC temperature in encoded motors and the cortex or power expander controlling them.
- Monitoring of PTC temperature and limiting of motor current if the maximum PTC is reached.
- Monitoring and limiting of average motor current if a preset current threshold is passed.
- Slew rate control of motors by controlling acceleration and deceleration.

This library is intended to be used by programmers who want to add another layer of intelligence to their software. It is probably not appropriate for beginners and some understanding of using functions and tasks in ROBOTC is a pre-requisite.

This software will not overcome the problems of a poorly designed and built robot. If the robot's motors are working outside of the safe design limits then, although this software can help, you will continue to have control issues. This software will help mostly with the following situations.

- Motor overheating during autonomous periods when unexpected events would cause a motor to stall.
- Motor overheating when in pushing matches with other robots.
- Motor overheating caused by enthusiastic driving under competition conditions..

Installation

The smart motor library is included in the main ROBOTC source file as follows

```
//*!!Code automatically generated by 'ROBOTC' configuration wizard !!*//
// Include smart motor library
#include "SmartMotorLib.c"
task main()
{
    // user code
}
```

The smart motor library can be located in the following directories on the PC

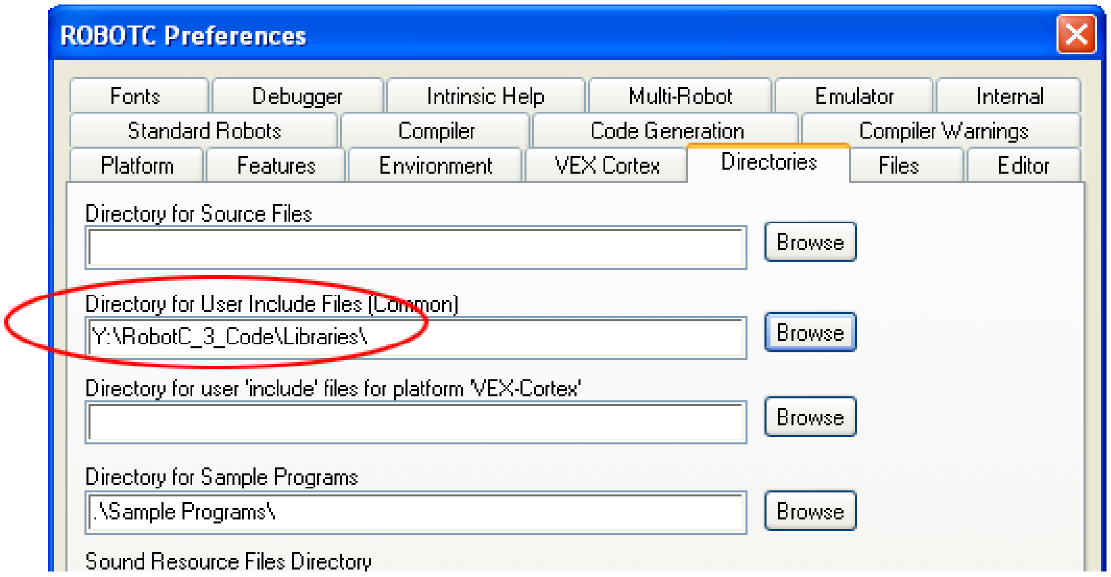
1. The same directory as the main source file, in this case the #include directive is as follows.

```
#include "SmartMotorLib.c"
```

2. A subdirectory in the directory of the main source file, for example in the "libraries" sub directory.

```
#include "libraries\SmartMotorLib.c"
```

3. In the path specified in the ROBOTC Directories preference pane



Directory for Source Files

Directory for User Include Files (Common)
Y:\RobotC_3_Code\Libraries\

Directory for user 'include' files for platform 'VEX-Cortex'

Directory for Sample Programs
.\Sample Programs\

Sound Resource Files Directory

How to use the library

The general procedure for using the smart motor library is as follows.

- Initialize the library
- Change default behavior and activate current limiting
- Start the tasks used by the library

Initialization

The smart motor library is initialized by calling the function

```
SmartMotorsInit();
```

This function initializes the data structures used by the library by detecting each type of motor available, what type of encoder, if any, is connected to the motor and where on the cortex the motor is installed.

Changing default behavior

The default behavior of the library is in a monitoring mode only, action is not taken if an over current or temperature condition occurs. The library also cannot detect the presence of a power expander or sharing of encoders by more than one motor, these conditions are setup as follows.

Using a power expander.

To enable use of a power expander, call the following function after initialization with the motors connected to the power expander as parameters. This example shows four motors being added.

```
SmartMotorsAddPowerExtender( port2, port3, port4, port5);
```

Linking motors

The smart motor library will work best when every motor is equipped with its own encoder; however, this is generally expensive and impractical for most competition robots. If multiple motors are powering, for example, an arm or lift system, then it is likely they will all be mechanically coupled and able to share one encoder. In this case they may be linked by using the following function.

```
SmartMotorLinkMotors( port2, port3 );
```

The first parameter, in this example port2, is the master motor, the motor that has the encoder specified in the “motors & sensors” dialog. The second parameter is the slave motor, this motor will share the encoder on the master motor. This function can be used multiple times as necessary; for example, if four motors are driving an arm and only one is encoded then three calls will be made.

```
SmartMotorLinkMotors( port2, port3 );
SmartMotorLinkMotors( port2, port4 );
SmartMotorLinkMotors( port2, port5 );
```

Change gear ratio

If the red quadrature encoder is being used with a motor it may not be on the motor drive axle but on a different axle that is connected by gears. If this is the case then it may not be turning at the same speed as the motors output. In this case the smart motor library needs to be given this information by using the following function.

```
SmartMotorsSetEncoderGearing( port2, 3.0 ) ;
```

The first parameter is the motor port number, the second parameter is the ratio of encoder revolutions to motor revolutions, in this example it is 3:1.

Enable temperature or current limit monitor

The smart motor library estimates the temperature of the thermal fuse in both the motor, cortex and power expander. If the library determines that a PTC is about to trip it can reduce the motors current, and therefore PTC temperature, by reducing the command value sent to it. This functionality is disabled as default and can be enabled by calling the function

```
SmartMotorPtcMonitorEnable();
```

As an alternative, the smart motor library can detect average current through the motor and reduce the command value if a preset threshold is reached. This functionality is also disabled after initialization and can be enabled by using this function.

```
SmartMotorCurrentMonitorEnable();
```

Only one of the two monitoring methods may be used, they cannot both be enabled.

Use the power expander’s status port

The power expander has an analog output that may be used to monitor battery voltage. As this monitoring is after the PTC in the power expander it can also be used to detect that the power expander’s PTC has tripped. The status port can also be monitored by the smart motor library, in this case the command value sent to the motors on the power expander is automatically reduced to zero to allow the PTC to reset. The status port is assigned to the smart motor library using the following function. This example assigns analog channel 1 to be the status port, it should have been configured as an analog input in the motors & sensors setup dialog.

```
SmartMotorSetPowerExpanderStatusPort( in1 );
```

Assign a LED to be used as status feedback

A digital output with installed LED may be used as status to show when the smart motor library has limited current on a cortex control bank (a control bank is a group of five motors sharing one PTC in the cortex) or any motor connected to that bank. A LED can be assigned for bank 0 on the cortex (ports 1-5), bank 1 on the cortex (ports 6-10) or the power expander. A single LED may be shared between banks or multiple LEDs may be used. This example assigns a different LED to be used as status for each control bank.

```
SmartMotorSetControllerStatusLed( SMLIB_CORTEX_PORT_0, dgtl10 );
SmartMotorSetControllerStatusLed( SMLIB_CORTEX_PORT_1, dgtl11 );
SmartMotorSetControllerStatusLed( SMLIB_PWREXP_PORT_0, dgtl12 );
```

Start motor tasks

After initialization and optionally changing default behavior the smart motor library tasks are started by calling this function.

```
SmartMotorRun();
```

This call will start two tasks, the first task calculates velocity for each motor and, based on this and the requested command value, the motors current and estimated PTC temperature. The second task deals with motor acceleration and deceleration and also checks for a limited command created by the current and PTC monitors..

The two tasks started by this function will run at a higher priority than a normal ROBOTC task, the speed and current calculation task is also set to be a higher priority than the slew rate and command limiting task as correct calculation of speed is the most important action of the library.

Avoid the use of hogCPU() function, this will severely impact the ability of the library to operate correctly.

Controlling Motors

Motors are controlled by using the SetMotor() function. This function must be used in place of the motor[] array that ROBOTC provides. For example, to set a motor speed to 127 use

```
SetMotor( port1, 127 );
```

Rather than

```
motor[ port1 ] = 127;
```

Using the standard motor[] array will not allow the smart motor library to limit commands to the motors. Use SetMotor anywhere you would normally use motor[].

Using the encoders

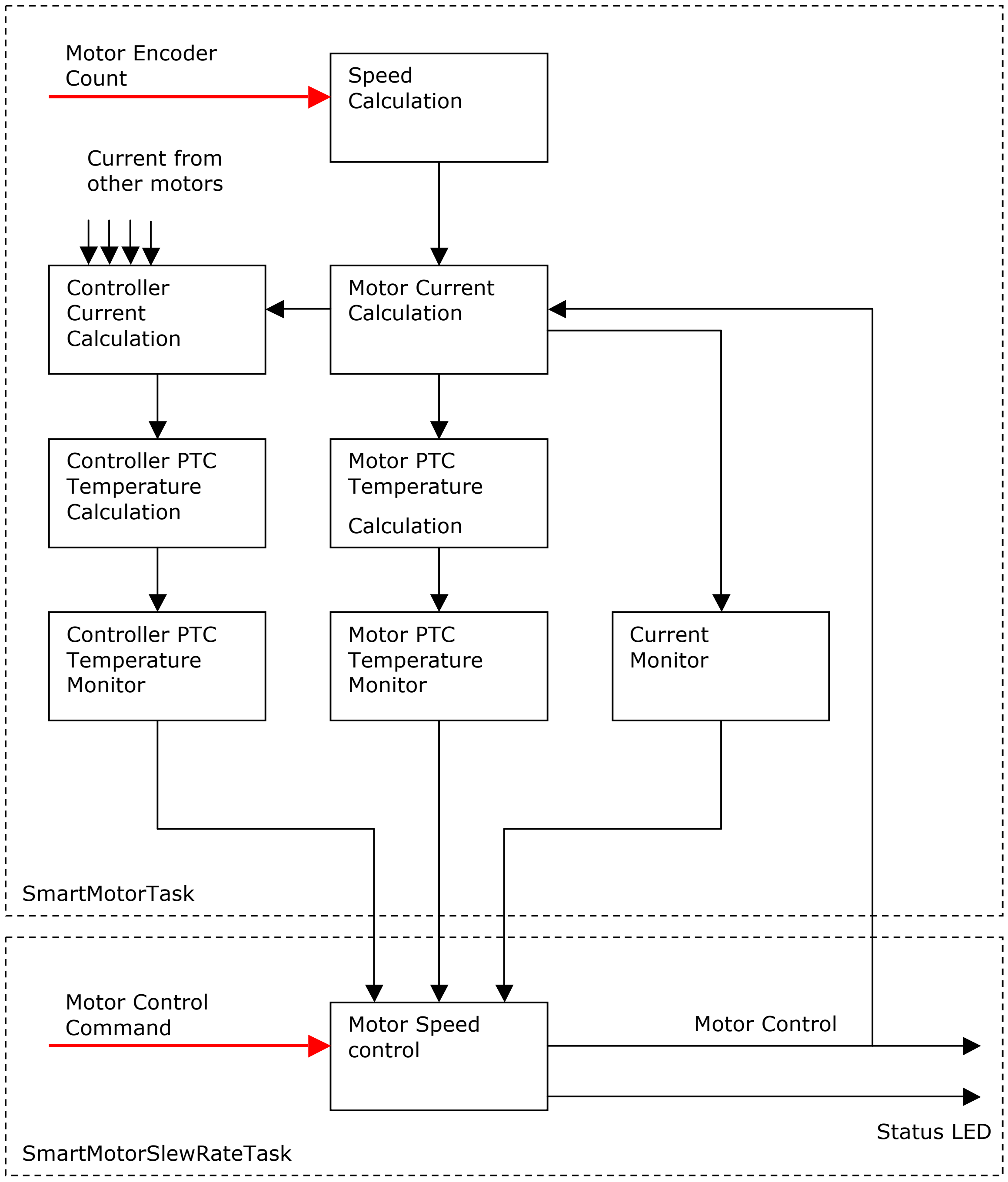
The smart motor library relies on the motor encoders for the calculation of velocity. A task samples the encoder every 100mS and calculates speed based on the change in encoder value. If the encoder is reset to 0 then a large change in value may be detected and an instantaneous high velocity calculated (it could also be smaller but it is most likely to be larger). Although the library does not try and clip this large value, it does limit the theoretical back emf that would be generated by this high velocity so as not to create impossible current transients.

The motor encoder values are obtained as 32 bit values and will not overflow in normal operation. If you want to be able to reset values to zero then using a wrapper function for obtaining values is the best solution, for example, these two functions could replace the normal calls to nMotorEncoder[].

```
long    nMotorEncoderOffsets[ 10 ] = {0,0,0,0,0,0,0,0,0,0};
void
EncoderSetValue( tMotor index, float value )
{
    // note, we pass value as float due to bug in ROBOTC V3.51

    // bounds check index
    if((index < 0) || (index >= kNumbOfTotalMotors))
        return;

    // calculate and save offset
    nMotorEncoderOffsets[ index ]  = nMotorEncoder[ index ] - value;
}
long
EncoderGetValue(tMotor index)
{
    // bounds check index
    if((index < 0) || (index >= kNumbOfTotalMotors))
        return(0);

    return( nMotorEncoder[ index ] - nMotorEncoderOffsets[ index ] );
}
```

Module structure

This gives an overview of how the library is structured. Inputs are the motor encoder values and the motor requested speeds.


Motor Encoder Count
Speed Calculation
Current from other motors
Controller Current Calculation
Motor Current Calculation
Controller PTC Temperature Calculation
Motor PTC Temperature Calculation
Controller PTC Temperature Monitor
Motor PTC Temperature Monitor
Current Monitor
SmartMotorTask
Motor Control Command
Motor Speed control
Motor Control
Status LED
SmartMotorSlewRateTask

Function Reference

Initialization functions

`void SmartMotorsInit()`

Initialize the smart motor library

This function initializes the various structures and variables used by the smart motor library. The motor type and presence of an encoder are read using ROBOTC functions. This function should be called once at the beginning of a program, the pre_auton() function in a competition template is a good place to put this call.

`void SmartMotorsAddPowerExtender( int p0, int p1, int p2, int p3 )`

Add a power expander as a smart controller

This function reassigns up to four motors to a power expander rather than ports on the cortex. Examples.

// the motor on port4 is assigned to the power expander

SmartMotorsAddPowerExtender(port4);

// three motors are assigned to the power expander

SmartMotorsAddPowerExtender(ArmMotorA, ArmMotorB, ArmMotorC);

`void SmartMotorLinkMotors( tMotor master, tMotor slave )`

Allows two motors to share the same encoder

This function assigns the encoder currently used by the master motor to also be used by the slave motor. This is useful if, for example, one motor has an IME and is mechanically coupled to another without an IME.

// port3 motor shares the encoder on port2

SmartMotorLinkMotors(port2, port3)

`void SmartMotorsSetEncoderGearing( tMotor index, float ratio )`

Allows for changing the relationship between an encoder and motors rotation.

This function is used for motors that are coupled to a quadrature encoder but with a gearing other than 1:1.

`void SmartMotorSetRpmSensor( tMotor index, tSensors port, float ticks_per_rev, bool reversed = false )`

This function can be used to assign any sensor to a motor for calculating rpm, the most likely situation would be a potentiometer. The parameter ticks_per_rev is the sensor change for one revolution of the motor, for example, a potentiometer with 1:1 gearing has about 6000 ticks_per_rev, if the gearing was 7:1 then this would be reduced to about 857. Set reversed to true if the potentiometer value decreases with positive motor command values.

Control functions

`void SmartMotorPtcMonitorEnable()`

`void SmartMotorPtcMonitorDisable()`

Functions to enable and disable the current limit based on PTC temperature. Current limit is disabled initially.

`void SmartMotorCurrentMonitorEnable()`

`void SmartMotorCurrentMonitorDisable()`

Functions to enable and disable the current limit based on a current threshold. Current limit is disabled initially. The default current limit is the PTC safe current, this can be changed on a per motor basis if needed.

`void SmartMotorSetFreeRpm( tMotor index, short max_rpm )`

This is a function to change the unloaded free running speed of the motor. The default free run speed is 120 rpm for the 269 motor and 110 rpm for the 393 motor. This function allows fine-tuning of the current calculations on a motor-by-motor basis.

`void SmartMotorSetLimitCurent ( tMotor index, float current)`

This is a function to change the current limit threshold. The default threshold is the PTC safe current for the motor.

`void  SmartMotorSetSlewRate( tMotor index, int slew_rate )`

This is a function to change the slew rate for the motor. The default slew rate is 10, this means that the motor commanded value can only change by 10 each time the controlling loop runs. The control loop acts every 15mS which therefore causes a full forward to reverse change (127 to -127) to take approximately 375mS (25 x 15mS). A smaller slew rate means slower acceleration or deceleration. A larger slew rate means faster acceleration or deceleration. A value of 255 essentially turns slew rate control off. My advice, leave it alone for thr drive system, perhaps make larger for low momentum applications like intakes.

`void  SmartMotorSetControllerStatusLed( int index, tSensors port )`

Associate a digital output with a controller to be used as an over current status LED. The port should be defined as a digital output or VEX led. Index can be one of the following

SMLIB_CORTEX_PORT_0

SMLIB_CORTEX_PORT_1

SMLIB_PWREXP_PORT_0

`void  SmartMotorRun()`

This starts the tasks used by the smart motor library and initiates the calculation of motor speed, current and PTC temperatures. Tasks are created with a higher than normal priority so standard user code will not stop them executing.

`void  SmartMotorStop()`

This stops the tasks used by the smart motor library. This is not used in a normal competition program.

Status functions

`float SmartMotorGetSpeed( tMotor index )`

Get the speed of the indexed motor, speed is returned in rpm as a float.

float SmartMotorGetCurrent(tMotor index)

float SmartMotorGetCurrent(tMotor index, int s)

Get the current of the indexed motor, current is returned in Amps as a float. An optional parameter set to non-zero returns current as a signed value rather than the default of unsigned.

```
// Get absolute current for the motor on port3
SmartMotorGetCurrent( port3 );

// Get current as a signed quantity for the motor on port3
SmartMotorGetCurrent( port3, 1 );
```

float SmartMotorGetTemperature (tMotor index)

Get the PTC temperature of the indexed motor, temperature is returned in degrees C as a float.

float SmartMotorGetControllerCurrent (short index)

Get the current of the indexed controller, curent is returned in Amps as a float. Index can be one of the following

SMLIB_CORTEX_PORT_0

SMLIB_CORTEX_PORT_1

SMLIB_PWREXP_PORT_0

float SmartMotorGetControllerTemperature (short index)

Get the PTC temperature of the indexed controller, temperature is returned in degrees C as a float. Index can be one of the following

SMLIB_CORTEX_PORT_0

SMLIB_CORTEX_PORT_1

SMLIB_PWREXP_PORT_0

void SmartMotorDebugStatus()

Dump some debug information to the debug stream.

Raw data access

```
smartMotor      *SmartMotorGetPtr( tMotor index )
```

Get a pointer to a smartMotor data structure. This allows access to all of the variables used by a single motor, it can be used to override defaults where there is not a dedicated function.

```
smartController *SmartMotorControllerGetPtr( short index )
```

Get a pointer to a smartController data structure. This allows access to all of the variables used by a single control bank, it can be used to override default where there is not a dedicated function.

Motor control

```
void  SetMotor( int index, int value = 0 )
```

This function must be used to set the required motor speed, it replaces use of the motor[] array in ROBOTC. For example, to set a motor speed to 127 use

```
SetMotor( port1, 127 );
```

Rather than

```
motor[ port1 ] = 127;
```

Function prototypes

```
// Initialization
void      SmartMotorsInit( void );
void      SmartMotorLinkMotors( tMotor master, tMotor slave );
void      SmartMotorsSetEncoderGearing( tMotor index, float ratio );
void      SmartMotorsAddPowerExtender( int p0, int p1, int p2, int p3 )

// Status
float     SmartMotorGetSpeed( tMotor index );
float     SmartMotorGetCurrent( tMotor index, int s );
float     SmartMotorGetTemperature( tMotor index );
int       SmartMotorGetLimitCmd( tMotor index );

float     SmartMotorGetControllerCurrent( short index );
float     SmartMotorGetControllerTemperature( short index );
void      SmartMotorDebugStatus( void );

// Control
void      SmartMotorPtcMonitorEnable( void );
void      SmartMotorPtcMonitorDisable( void );
void      SmartMotorCurrentMonitorEnable( void );
void      SmartMotorCurrentMonitorDisable( void );
void      SmartMotorSetLimitCurent( tMotor index, float current );
void      SmartMotorSetFreeRpm( tMotor index, short max_rpm );
void      SmartMotorSetSlewRate( tMotor index, int slew_rate );
void      SmartMotorRun( void );
void      SmartMotorStop( void );
void      SmartMotorSetControllerStatusLed( int index, tSensors port );

void      SetMotor( int index, int value = 0 );

// Access raw data
smartMotor      *SmartMotorGetPtr( tMotor index );
smartController *SmartMotorControllerGetPtr( short index );
```

```
// Private functions shown for reference - do not call from user code
void     SmartMotorSpeed( smartMotor *m, int deltaTime );
void     SmartMotorSimulateSpeed( smartMotor *m );
float    SmartMotorCurrent( smartMotor *m, float v_battery  );
float    SmartMotorControllerCurrent( smartController *s );
int      SmartMotorSafeCommand( smartMotor *m, float v_battery  );
float    SmartMotorTemperature( smartMotor *m, int deltaTime );
float    SmartMotorControllerTemperature( smartController *s, int deltaTime  );
void     SmartMotorMonitorPtc( smartMotor *m, float v_battery );
void     SmartMotorControllerMonitorPtc( smartController *s, float v_battery );
void     SmartMotorMonitorCurrent( smartMotor *m, float v_battery );
void     SmartMotorControllerSetLed( smartController *s );

// tasks
task     SmartMotorTask( void );
task     SmartMotorSlewRateTask( void );
```

Examples

Example 1

Simple 4 motor arcade drive with 4 encoded motors

```
#pragma config(I2C_Usage, I2C1, i2cSensors)
#pragma config(Sensor, I2C_1,  EncRF,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_2,  EncLF,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_3,  EncLB,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_4,  EncRB,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Motor,  port2,  MotorRF,   tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C_1, 1000)
#pragma config(Motor,  port3,  MotorLF,   tmotorVex393, openLoop,           encoder, encoderPort, I2C_2, 1000)
#pragma config(Motor,  port8,  MotorLB,   tmotorVex393, openLoop,           encoder, encoderPort, I2C_3, 1000)
#pragma config(Motor,  port9,  MotorRB,   tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C_4, 1000)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

#include "Libraries\SmartMotorLib.c"

task main()
{
    long    drive_l_motor;
    long    drive_r_motor;
    // Initialize the Smart Motor Library
    SmartMotorsInit();
    // Limit current based on default motor current threshold
    SmartMotorCurrentMonitorEnable();
    // Run smart motors
    SmartMotorRun();

    while(1)
        {
        // drive
        drive_l_motor = vexRT[ Ch3 ] + vexRT[ Ch4 ];
        drive_r_motor = vexRT[ Ch3 ] - vexRT[ Ch4 ];
        SetMotor( MotorLF, drive_l_motor);
        SetMotor( MotorLB, drive_l_motor);
        SetMotor( MotorRF, drive_r_motor);
        SetMotor( MotorRB, drive_r_motor);
        wait1Msec(25);
        }
}
```

Example 2

Display speed of two encoded motors

No current limiting

```
#pragma config(I2C_Usage, I2C1, i2cSensors)
#pragma config(Sensor, I2C_1,  EncRF,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_2,  EncLF,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Motor,  port2,  MotorRF,   tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C_1, 1000)
#pragma config(Motor,  port3,  MotorLF,   tmotorVex393, openLoop,           encoder, encoderPort, I2C_2, 1000)
#pragma config(Motor,  port8,  MotorLB,   tmotorVex393, openLoop)
#pragma config(Motor,  port9,  MotorRB,   tmotorVex393, openLoop, reversed)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

#include "Libraries\SmartMotorLib.c"

task main()
{
    long    drive_l_motor;
    long    drive_r_motor;
    string  str;

    // Initialize the Smart Motor Library
    SmartMotorsInit();
    // Run smart motors
    SmartMotorRun();
    while(1)
        {
        // drive
        drive_l_motor = vexRT[ Ch3 ] + vexRT[ Ch4 ];
        drive_r_motor = vexRT[ Ch3 ] - vexRT[ Ch4 ];
        SetMotor( MotorLF, drive_l_motor);
        SetMotor( MotorLB, drive_l_motor);
        SetMotor( MotorRF, drive_r_motor);
        SetMotor( MotorRB, drive_r_motor);

        sprintf( str, "%7.2f  %7.2f", SmartMotorGetSpeed( MotorLF ), SmartMotorGetSpeed( MotorRF ));
        displayLCDString(0, 0, str);

        wait1Msec(25);
        }
}
```

Example 3

4 encoded drive motors

4 Arm motors sharing one quad encoder on digital input 1&2

2 non-encoded intake motors

1 Power expander

```
#pragma config(I2C_Usage, I2C1, i2cSensors)
#pragma config(Sensor, dgtl1,  EncArm,    sensorQuadEncoder)
#pragma config(Sensor, dgtl12, statusLED, sensorLEDtoVCC)
#pragma config(Sensor, I2C_1,  EncRF,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_2,  EncLF,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_3,  EncLB,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_4,  EncRB,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Motor,  port1,  IntakeL,   tmotorVex269, openLoop)
#pragma config(Motor,  port2,  MotorRF,   tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C_1, 1000)
#pragma config(Motor,  port3,  MotorLF,   tmotorVex393, openLoop,           encoder, encoderPort, I2C_2, 1000)
#pragma config(Motor,  port4,  MotorArm1, tmotorVex393, openLoop,           encoder, encoderPort, dgtl1, 1000)
#pragma config(Motor,  port5,  MotorArm2, tmotorVex393, openLoop)
#pragma config(Motor,  port6,  MotorArm3, tmotorVex393, openLoop)
#pragma config(Motor,  port7,  MotorArm4, tmotorVex393, openLoop)
#pragma config(Motor,  port8,  MotorLB,   tmotorVex393, openLoop,           encoder, encoderPort, I2C_3, 1000)
#pragma config(Motor,  port9,  MotorRB,   tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C_4, 1000)
#pragma config(Motor,  port10, IntakeR,   tmotorVex269, openLoop, reversed)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

#include "Libraries\SmartMotorLib.c"
```

```
task main()
{
    short   arm_drive;
    short   intake_drive;
    long    drive_l_motor, drive_r_motor;

    // Initialize the Smart Motor Library
    SmartMotorsInit();
    // Arm motors on power extender
    SmartMotorsAddPowerExtender( MotorArm1, MotorArm2, MotorArm3, MotorArm4 );
    // one encoder for all arm motors
    SmartMotorLinkMotors( MotorArm1, MotorArm2 );
    SmartMotorLinkMotors( MotorArm1, MotorArm3 );
    SmartMotorLinkMotors( MotorArm1, MotorArm4 );
    // Limit current in the motors using PTC temperatures
    SmartMotorPtcMonitorEnable();
    // led to monitor the power expander
    SmartMotorSetControllerStatusLed( SMLIB_PWREXP_PORT_0, statusLED );

    // Run smart motors
    SmartMotorRun();

    while(1) {
        // drive
        drive_l_motor = vexRT[ Ch3 ] + vexRT[ Ch4 ];
        drive_r_motor = vexRT[ Ch3 ] - vexRT[ Ch4 ];
        SetMotor( MotorLF, drive_l_motor);
        SetMotor( MotorLB, drive_l_motor);
        SetMotor( MotorRF, drive_r_motor);
        SetMotor( MotorRB, drive_r_motor);

        // arm
        arm_drive = vexRT[ Ch2 ];
        SetMotor( MotorArm1, arm_drive );
        SetMotor( MotorArm2, arm_drive );
        SetMotor( MotorArm3, arm_drive );
        SetMotor( MotorArm4, arm_drive );

        // intake
        if( vexRT[ Btn6U ] )
            intake_drive = 100;
        else
        if( vexRT[ Btn6D ] )
            intake_drive = -100;
        else
            intake_drive = 0;

        SetMotor( IntakeL, intake_drive );
        SetMotor( IntakeR, intake_drive );

        wait1Msec(25);
        }
}
```

Example 4

Simple 4 motor arcade drive with 4 encoded motors (same as example 1) in the standard competition template.

```
#pragma config(I2C_Usage, I2C1, i2cSensors)
#pragma config(Sensor, I2C_1,  EncRF,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_2,  EncLF,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_3,  EncLB,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Sensor, I2C_4,  EncRB,     sensorQuadEncoderOnI2CPort,, AutoAssign)
#pragma config(Motor,  port2,  MotorRF,   tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C_1, 1000)
#pragma config(Motor,  port3,  MotorLF,   tmotorVex393, openLoop,           encoder, encoderPort, I2C_2, 1000)
#pragma config(Motor,  port8,  MotorLB,   tmotorVex393, openLoop,           encoder, encoderPort, I2C_3, 1000)
#pragma config(Motor,  port9,  MotorRB,   tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C_4, 1000)
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

#include "Libraries\SmartMotorLib.c"

#pragma platform(VEX)

//Competition Control and Duration Settings
#pragma competitionControl(Competition)
#pragma autonomousDuration(15)
#pragma userControlDuration(105)

#include "Vex_Competition_Includes.c"   //Main competition background code...do not modify!

/////////////////////////////////////////////////////////////////////////////////////////
```

```
//
//                          Pre-Autonomous Functions
//
// You may want to perform some actions before the competition starts. Do them in the
// following function.
//
/////////////////////////////////////////////////////////////////////////////////////////

void pre_auton()
{
    // Set bStopTasksBetweenModes to false if you want to keep user created tasks running between
    // Autonomous and Tele-Op modes. You will need to manage all user created tasks if set to false.
    bStopTasksBetweenModes = false;

    // Initialize the Smart Motor Library
    SmartMotorsInit();
    // Limit current based on default motor current threshold
    SmartMotorCurrentMonitorEnable();
    // Run smart motors
    SmartMotorRun();
}

/////////////////////////////////////////////////////////////////////////////////////////
//
//                                 Autonomous Task
//
// This task is used to control your robot during the autonomous phase of a VEX Competition.
// You must modify the code to add your own robot specific commands here.
//
/////////////////////////////////////////////////////////////////////////////////////////

task autonomous()
{
    // in case we were running
    StopTask( usercontrol );

    // Silly automonous, drive for 5 seconds
    SetMotor( MotorLF, 127 );
    SetMotor( MotorLB, 127);
    SetMotor( MotorRF, 127);
    SetMotor( MotorRB, 127);

    wait1Msec( 5000 );

    SetMotor( MotorLF, 0 );
    SetMotor( MotorLB, 0);
    SetMotor( MotorRF, 0);
    SetMotor( MotorRB, 0);
}

/////////////////////////////////////////////////////////////////////////////////////////
//
//                                  User Control Task
//
```

```
// This task is used to control your robot during the user control phase of a VEX Competition.
// You must modify the code to add your own robot specific commands here.
//
/////////////////////////////////////////////////////////////////////////////////////////

task usercontrol()
{
    long    drive_l_motor;
    long    drive_r_motor;

    // in case we were running
    StopTask( autonomous );

    while(1)
        {
        // drive
        drive_l_motor = vexRT[ Ch3 ] + vexRT[ Ch4 ];
        drive_r_motor = vexRT[ Ch3 ] - vexRT[ Ch4 ];
        SetMotor( MotorLF, drive_l_motor);
        SetMotor( MotorLB, drive_l_motor);
        SetMotor( MotorRF, drive_r_motor);
        SetMotor( MotorRB, drive_r_motor);
        wait1Msec(25);
        }
}
```