File: C:\Users\Terry\SkyDrive\Documents\Vex\RoboBowl\RoboBowlTeleopDemo.c

```
#pragma config(I2C Usage, I2C1, i2cSensors)
#pragma config(Sensor, in1,    armPotentiometer, sensorPotentiometer)
#pragma config(Sensor, dgtl1,  rightRearBumper, sensorTouch)
#pragma config(Sensor, dgtl2,  leftRearBumper, sensorTouch)
#pragma config(Sensor, dgtl3,  sonar,          sensorSONAR cm)
#pragma config(Sensor, dgtl5,  clawSwitch,     sensorTouch)
#pragma config(Sensor, dgtl6,  rightYellow,    sensorLEDtoVCC)
#pragma config(Sensor, dgtl7,  rightGreen,     sensorLEDtoVCC)
#pragma config(Sensor, dgtl8,  rightRed,       sensorLEDtoVCC)
#pragma config(Sensor, dgtl10, leftGreen,      sensorLEDtoVCC)
#pragma config(Sensor, dgtl11, leftRed,        sensorLEDtoVCC)
#pragma config(Sensor, dgtl12, leftYellow,     sensorLEDtoVCC)
#pragma config(Sensor, I2C_1,  leftEncoder,    sensorQuadEncoderOnI2CPort,    , AutoAssign )
#pragma config(Sensor, I2C_2,  rightEncoder,   sensorQuadEncoderOnI2CPort,    , AutoAssign )
#pragma config(Motor,  port1,           leftMotor,     tmotorVex393_HBridge, openLoop, reversed, driveLeft, en
#pragma config(Motor,  port6,           clawMotor,     tmotorVex393_MC29, openLoop, reversed)
#pragma config(Motor,  port7,           armMotor,      tmotorVex393_MC29, openLoop)
#pragma config(Motor,  port10,          rightMotor,    tmotorVex393_HBridge, openLoop, driveRight, encoderPort
//*!!Code automatically generated by 'ROBOTC' configuration wizard               !!*//

#include "Utils.c"

/*
This demo shows a layer of automation on top of a teleop program.

The regular Arcade Drive mode is used for mobility, with the addition
of two safeties:
- The sonar is used to prevent forward collisions
- The rear bumpers are used to stop running the motors when backed
against the wall

The arm is automated to assist the grabbing and carrying of objects.
(See the armAction comment block for more details.)

Techniques demonstrated:
- PID position control of arm
- Live PID tuning of a running system
- passive monitoring of encoders with LED display
- state machine for sequencing actions concurrently with driving
- sonar and bump sensor based wall avoidance
```

```
*/

const int maxArmPower = 127;  // maximum power to the arm
const int clawHighPower = 60; // quickly close and open claw
const int clawGripPower = 10; // gently hold the object
const long clawTime = 500;    // ms to run claw for full open or close
const short noBumpCm = 10;    // minimum safe distance for sonar

const int armLow = 4050;  // potentiometer value when arm is ready to grab object
const int armHigh = 3600; // potentiometer value when arm is holding object
int armTarget = armLow;
int armPos;

// PID variables are global so they're visible in the debugger.
float Kp = 0.6;
float Ti = 200.0;
float Td = 40.0;
float iErrZone = 30.0;

int pidError;
float dT, dE;
float iE = 0;
int pidPower;

task armPID()
{
  int lastE = 0;
  long lastTime = time1[T1];

  while(true) {
    sleep(5); // make sure dT is never 0.
    long now = time1[T1];
    dT = now - lastTime;
    lastTime = now;

    // Calculate current error
    armPos = SensorValue[armPotentiometer];
    pidError = armTarget - armPos;

    // Calculate integral and differential terms
    iE = iabs(pidError) > iErrZone ? 0 : iE + pidError * dT;
```

```
    dE = (pidError - lastE) / dT;
    lastE = pidError;

    // PID formula: Standard Form
    // https://en.wikipedia.org/wiki/PID_controller#Ideal versus standard PID form
    pidPower = Kp * (pidError + iE/Ti + Td*dE);

    // Apply power to motor
    motor[armMotor] = symclip(pidPower, maxArmPower);
  }
}

/*
This section enables live-tuning of the arm PID values.
Use the left buttons to select the mode and tweak the values.
It's most helpful to be connected to the debugger while
tweaking so you can see what values you've selected.

Select the mode:  (tap button to select)
- Button 7 right = off (lock adjustments)
- Button 7 up = select Kp
- Button 7 left = select Ti
- Button 7 down = select Td

Adjust the value:  (tap button to single-step the value)
- Button 5 up/down adjusts the value up/down
*/
enum TuneMode { tmOff, tmKp, tmTi, tmTd };
TuneMode tuneMode = tmOff;

void tunePid()
{
  // Check for new mode selection
  if(vexRT[Btn7R])
    tuneMode = tmOff;
  else if(vexRT[Btn7U])
    tuneMode = tmKp;
  else if(vexRT[Btn7L])
    tuneMode = tmTi;
  else if(vexRT[Btn7D])
    tuneMode = tmTd;
```

```
  // Check for value adjustment
  float tuneDir = 0.0;
  static float tuneDirLast = 0.0;
  if(vexRT[Btn5U])
    tuneDir = 1.0;
  else if(vexRT[Btn5D])
    tuneDir = -1.0;

  // Allow single step per button press
  if(tuneDir == tuneDirLast)
    tuneDir = 0.0;
  else
    tuneDirLast = tuneDir;

  // Apply adjustment to selected value
  switch(tuneMode) {
  case tmOff:
    break;
  case tmKp:
    Kp += 0.1 * tuneDir;
    break;
  case tmTi:
    Ti += 10.0 * tuneDir;
    if(Ti <= 0.0) // Ti cannot be zero
      Ti = 10.0;
    break;
  case tmTd:
    Td += 1.0 * tuneDir;
    break;
  }
}

int leftEnc()
{
  return nMotorEncoder[leftMotor];
}

int rightEnc()
{
  return nMotorEncoder[rightMotor];
```

```
}

/*
Passively monitor the drive wheel encoders and light up
the red and green LEDs.
This is useful for detecting faults in the encoders.
*/
task wheelMonitor()
{
  while (true) {
    int lEnc = leftEnc();
    int rEnc = rightEnc();

    static int lLastEnc = 0;
    static int rLastEnc = 0;

    int lChange = lEnc - lLastEnc;
    int rChange = rEnc - rLastEnc;

    lLastEnc = lEnc;
    rLastEnc = rEnc;

    SensorValue[leftGreen] = lChange > 0;
    SensorValue[leftRed] = lChange < 0;
    SensorValue[rightGreen] = rChange > 0;
    SensorValue[rightRed] = rChange < 0;

    sleep(50);
  }
}

/*
Automate the grabbing and carrying of objects.

Behavior:
- Normally hold the arm low and claw open.
- When an object is detected in the claw (or Btn6U is pressed),
grab and raise the arm.
- When the down button (Btn6D) is pressed,
lower the arm and release the object.
```

File: C:\Users\Terry\SkyDrive\Documents\Vex\RoboBowl\RoboBowlTeleopDemo.c

```
Implementation:
- A state machine is used to sequence the grabbing and raising of the arm.
- Each state will transition to another state on the correct trigger.
- The states are: Ready, Grabbing, Raising, Holding, Lowering, Releasing
- Grabbing and Releasing require the claw motor to run for a minimum
amount of time (clawTime)
- Raising and lowering require the arm to arrive near the target position
- If the master request (grabIt) changes mid-move,
then the sequence is reversed appropriately.
*/
enum GrabState { gsReady, gsGrabbing, gsRaising, gsHolding, gsLowering, gsReleasing };
GrabState grabState = gsLowering;

void armAction()
{
  bool grabRequest = vexRT[Btn6U] || SensorValue[clawSwitch] == true;
  bool releaseRequest = !!vexRT[Btn6D]; // !! converts int -> bool
  // making grabIt static allows triggers to be sticky
  static bool grabIt = false;
  if(grabRequest)
    grabIt = true;
  if(releaseRequest) // release after grab gives release priority
    grabIt = false;

  long now = time1[T1];
  static long then = now;
  long since = now - then;

  switch(grabState) {
  case gsReady:
    if(grabIt)
    {
      grabState = gsGrabbing;
      then = now;
    }
    break;

  case gsGrabbing:
    if(!grabIt)
    {
      grabState = gsReleasing;
```

```
      then = now;
    }
    else if(since < clawTime)
    {
      motor[clawMotor] = clawHighPower;
    }
    else
    {
      motor[clawMotor] = clawGripPower;
      armTarget = armHigh;
      grabState = gsRaising;
    }
    break;

  case gsRaising:
    if(!grabIt)
    {
      armTarget = armLow;
      grabState = gsLowering;
    }
    else if(iabs(armHigh - armPos) < 50)
      grabState = gsHolding;
    break;

  case gsHolding:
    if(!grabIt)
    {
      armTarget = armLow;
      grabState = gsLowering;
    }
    break;

  case gsLowering:
    if(grabIt)
    {
      armTarget = armHigh;
      grabState = gsRaising;
    }
    else if(iabs(armLow - armPos) < 50)
    {
      grabState = gsReleasing;
```

```
      then = now;
    }
    break;

  case gsReleasing:
    if(grabIt)
    {
      grabState = gsGrabbing;
      then = now;
    }
    else if(since < clawTime)
    {
      motor[clawMotor] = -clawHighPower;
    }
    else
    {
      motor[clawMotor] = 0;
      grabState = gsReady;
    }
    break;
  }
}

bool leftBump()
{
  return SensorValue[leftRearBumper] == true;
}

bool rightBump()
{
  return SensorValue[rightRearBumper] == true;
}

unsigned short getSonar()
{
  return SensorValue[sonar];
}

void arcadeDrive()
{
  // Get user requests
```

```
  int moveRequest = deadzone(vexRT[Ch2]);
  int turnRequest = deadzone(vexRT[Ch1]);

  // Anti-collision limits
  if(moveRequest > 0 && getSonar() < noBumpCm)
  {
    moveRequest = 0;
  }
  if(moveRequest < 0 && (leftBump() || rightBump()))
  {
    moveRequest = 0;
  }

  motor[leftMotor]  = (moveRequest + turnRequest)/2;
  motor[rightMotor] = (moveRequest - turnRequest)/2;
}

/*
The main() task runs in a constant loop servicing requests
for the drive motors, the arm action, and the PID tuning.
Each of the three activities happen very quickly, so a
delay of 50 milliseconds gives the CPU time to work on the
other tasks before checking for user input again.
*/
task main()
{
  startTask(armPID);
  startTask(wheelMonitor);

  while(true)
  {
    arcadeDrive();
    armAction();
    tunePid();

    sleep(50);
  }
}
```