Includes special tips and tricks to make your robot more “SELF AWARE”

by Karanvir Panesar
Table of Contents

Pneumatics
1. Setting up the Solenoid Port.............................................................................................................3
2. Giving commands to solenoid.........................................................................................................5
3. Programming a Pneumatic Lift........................................................................................................6

Special Functions
5. The Straight Function....................................................................................................................9
6. Stall Detection..................................................................................................................................10
Programming pneumatics in RobotC is very easy. As you may already know, pneumatic pistons are controlled by solenoids. I will use RobotC to demonstrate how to send out a signal to the solenoid. Let’s assume that we have one solenoid with a pneumatic piston connected to it. In order to control the piston, we must send a digital signal. To set up the ports, open up the “Motors and Sensors Setup” dialog box by clicking Robot → Motors and Sensors Setup and then navigate to the “VEX 2.0 Digital Sensors 1-12” tab.
Find an open digital port and name your solenoid whatever you like. Since we need to send a digital signal to the solenoid, choose “Digital Out” as the Sensor type. Now, the solenoid is ready to be given command.
Giving command to the Solenoid

Programming pneumatics is a lot easier than it might seem. In fact, it is as easy as sending a 0 or a 1 to the solenoid. As you can guess, a value of 0 disengages the piston while a value of 1 engages it. In RobotC, you can access the solenoid using the “SensorValue[]” command as follows:

```
58  task usercontrol()
59  {
60    while (True)
61    {
62      SensorValue[solenoid] = 0;  //Disengages the piston
63      wait1Msec(1000);           //Waits 1 second
64      SensorValue[solenoid] = 1; //Engages the piston
65    }
66  }
```

And that is all the basic information you have to know in order to program pneumatics! In the next section, we will take a look at how to fully program a four-stage pneumatic lift.
The 4-stage pneumatic lift that we will be programming consists of six pistons and four solenoids. Two solenoids, which are wi-cabled, control the pistons that lift up while the other two, also wi-cabled, control the pistons that pull down.
The lift operates as follows:

→ All pistons engaged: Lift goes to 30 inches
→ Top pistons engaged: Lift goes to 11 inches
→ Bottom pistons engaged: Lift goes to 20 inches
→ No pistons engaged: Lift goes to Floor height

To control the different heights, I will be using four buttons on the VEX Joystick channels 5 and 6.

First, define the buttons in RobotC as follows:

```c
void Joystick()
{
    joy1_Ch2       = vexRT[Ch2];
    joy1_Ch3       = vexRT[Ch3];
    joy1LiftBottom = vexRT[Btn5D];
    joy1LiftTop    = vexRT[BtnSU];
    joy1ManipIn    = vexRT[Btn6U];
    joy1ManipOut   = vexRT[Btn6D];
    halfIntake     = vexRT[Btn8R];
    joy2_Ch3       = vexRT[Ch3Xmtr2];
    joy2_Ch2       = vexRT[Ch2Xmtr2];
    joy2_mid_goal  = vexRT[Btn6DXmtr2];
    joy2_high_goal = vexRT[Btn6UXmtr2];
    joy2_low_goal  = vexRT[Btn5UXmtr2];
    joy2_floor_goal = vexRT[Btn5DXmtr2];
    joyToggle      = vexRT[Btn7U];
}
void TankDrive ()
{
    // Right Motor Controls
    if (joy1_Ch2 <= DEAD_ZONE && joy1_Ch2 >= -DEAD_ZONE)
        rFwd = NEUTRAL;
    else if(joy1_Ch2 > DEAD_ZONE)
After setting up the buttons, programming the lift is as easy as checking if a button is pressed and then engaging the right pistons.

```c
void Lift()
{
    if(joy2_high_goal == 1){
        SensorValue[solenoidTop] = 1;
        SensorValue[solenoidBottom] = 1;
    }
    else if (joy2_mid_goal == 1){
        SensorValue[solenoidTop] = 0;
        SensorValue[solenoidBottom] = 1;
    }
    else if (joy2_low_goal == 1){
        SensorValue[solenoidTop] = 1;
        SensorValue[solenoidBottom] = 0;
    }
    else if (joy2_floor_goal == 1){
        SensorValue[solenoidTop] = 0;
        SensorValue[solenoidBottom] = 0;
    }
}
```

This function called “Lift” could be called during the User Control period as follows:

```c
task usercontrol()
{
    while (true)
    {
        Joystick();
        Lift();
        TankDrive ();
        Intake ();
        SetDriveMotors();
        SetManipulatorMotors();
    }
}
```

Now you are ready to program your own functions to control VEX Pneumatics!
If you have faced the problem of your robot moving in curved path even if you set both sides of the drive to go at the same speed, there is a quick solution that will make the robot move in an almost perfectly straight line. The trick to moving straight in autonomous is to actively monitor the encoder values as the robot moves forward. If the left encoder value exceeds that of the right one, then you would reduce the motor speed on the left side and if the right encoder value is higher than the left one, you would reduce the motor speed on the right side. Lastly, if both encoder values are equal, it is optimal to set motors on both sides of the drive at equal speed. You can implement this idea into your code as follows:

```c
void Straight(int distance, int drive_speed){
    int direction = distance/abs(distance);
    distance = abs(distance);

    Reset_Sensors();

    while (abs(SensorValue[lt_enc]) < distance && abs(SensorValue[rt_enc]) < distance){
        if (SensorValue[lt_enc] > SensorValue[rt_enc]){  
            Lfwd = (drive_speed-10)*direction;
            rfwd = drive_speed*direction;
            SetDriveMotors();
        } else {  
            Lfwd = drive_speed*direction;
            rfwd = (drive_speed+10)*direction;
            SetDriveMotors();
        }
        if (SensorValue[lt_enc] == SensorValue[rt_enc]){  
            Lfwd = drive_speed*direction;
            rfwd = drive_speed*direction;
            SetDriveMotors();
        }
        Stop(ROBOT);
    }
}
```

This function takes distance and speed as arguments and uses those to move the robot in a straight line. A negative distance results in the robot moving backwards since the motor speed is multiplied by direction, which would be -1 in this case. As you can see, the code automatically adjusts the motor speeds based on the encoder values and consequently makes the robot move in a straight line.
Sometimes during the autonomous period, the robot goes forward too much and slams into a wall or a goal, hence stalling the motors. This could be prevented automatically by the code through actively monitoring the encoder values. While the robot is commanded to be moving, if the encoder values do not change over a certain period of time (around 500ms) then it means that the robot is being blocked by an object and is stalling its motors. Since we should only check for stalling motors when the robot is supposed to be moving, the **CheckStall** function could be put directly in the **Straight** function mentioned in the previous lesson.

To program the stall detection function, we must start a timer and compare the encoder value at the start of the timer with the encoder value at the end of our time tolerance. The following code shows the **CheckStall** function comparing \( \text{last_pos} \) (the encoder value at time = 0ms) with \( \text{current_pos} \) (the encoder value at time = 500ms):

\[
*\text{TOLERANCE} = 50
\]

```c
void CheckStall(int direction)
{
    if(direction > 0){
        if(current_pos < (last_pos + TOLERANCE))
            stalling = true;
        else
            stalling = false;
    }
    else if(direction < 0){
        if(current_pos > (last_pos - TOLERANCE))
            stalling = true;
        else
            stalling = false;
    }
}
```

The **CheckStall** function can be now used within the **Straight** function as follows:

```c
void Straight(int distance, int drive_speed){
    int direction = distance / abs(distance);
    distance = abs(distance);

    last_pos = -1;
    current_pos = 0;
    stalling = false;
    Reset_Sensors();
    ClearTimer(T1);
    while (abs(SensorValue[lt_enc]) < distance && abs(SensorValue[rt_enc]) < distance && !stalling) {
        if(time[T1] > STALL TOLERANCE){
            CheckStall(direction);
            last_pos = current_pos;
            ClearTimer(T1);
        }
        if (SensorValue[lt_enc] > SensorValue[rt_enc]) {
            Lfwd = (drive_speed - 10)*direction;
            Lfwd = drive_speed*direction;
        }
```
With this code, every time the timer exceeds the tolerance (500ms worked fine for our robot), the `CheckStall` function is called. If the function determines that the encoder values have not changed enough within the last 500ms, then it sets the boolean named `stalling` to true. Otherwise, `stalling` remains false and the while statement continues to loop through. Note that after the `CheckStall` function is called, the `last_pos` is set equal to the `current_pos` and the timer is cleared back to 0ms. This is so that when the time equals 500ms again, the `CheckStall` function can compare a new `current_pos` with the encoder value of the robot 500ms ago. This cycle repeats itself until the robot reaches its destination or it stalls. Now even your robot has some sense of what it is doing.