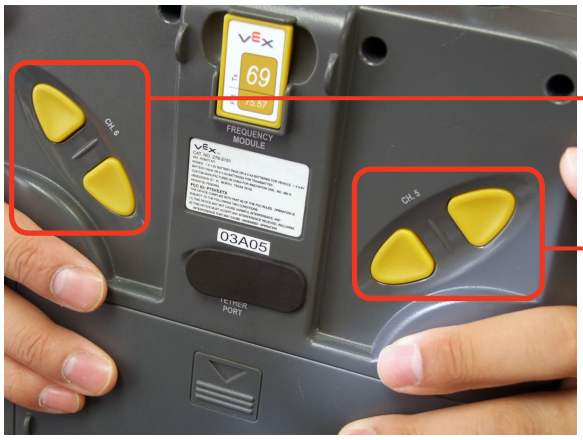
Buttons Transmitter Buttons

In this lesson, you will learn how the buttons on the back of the Vex Transmitter can be used to send values to the robot.

You already know how the **joysticks** on the front of the Transmitter work. Let's take a look at the **buttons** on the back.

There are four buttons on the back side of the Vex Transmitter, divided into two groups of two buttons. As with the joysticks, the point of having buttons on the Transmitter is to communicate a **value** to the robot. Recall that this is done by sending those values over radio **channels**. Each set of buttons, two on the left and two on the right, is tied to one radio channel.



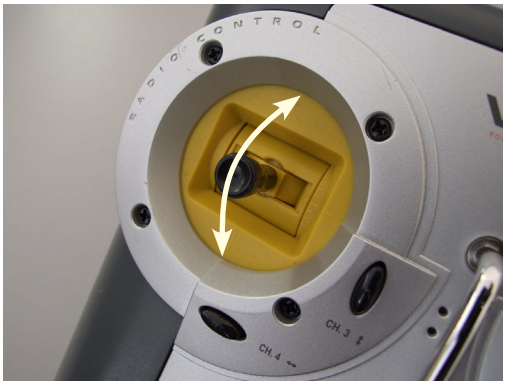
Channel 6
The two buttons operated by the right hand (remember, the buttons are on the back of the Transmitter) will control the value sent on Radio Channel 6.

Channel 5
The two buttons operated by the left hand will control the value sent on Radio Channel 5.

With the joysticks, the values were tied to the position of the sticks along certain directional axes. Therefore, any value in the range –127 to 127 could be sent. Buttons, however, are either pushed or not pushed, and so only a few specific values can be sent.

© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems

Buttons Transmitter Buttons *(cont.)*



Joystick
Values in the range –127 to 127 can be generated on the channel, depending on the exact position of the stick along the axis.

Buttons
Exact values of 127 or –127 can be generated on the channel when one of the two buttons is pushed in. A value of 0 is sent instead, if neither or both buttons are pushed.

End of Section

A button press on the Transmitter changes the value on the corresponding Radio Control channel (Ch. 5 for the left-hand buttons, Ch. 6 for the right-hand ones). By default, a value of 0 (zero) is sent when nothing is pressed. Pushing the top button changes that value to 127 as long as the button is held. Pushing the bottom button instead gives a value of –127. If both buttons on the same channel are pressed, they cancel out and a 0 is sent again.

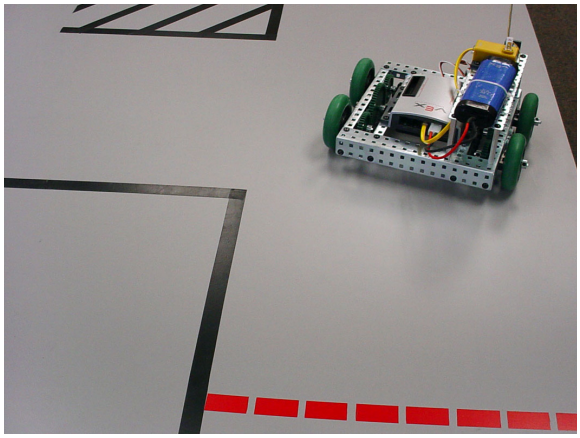
In the next section, you will learn how to make the robot respond to a Transmitter button press. The program will watch and respond to values on Radio channels 5 and 6 (left and right button groupings).

© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems

Buttons Remote Start

In this lesson, you will learn how to use input from the buttons on the back of the Transmitter to give you control when your robot "starts".

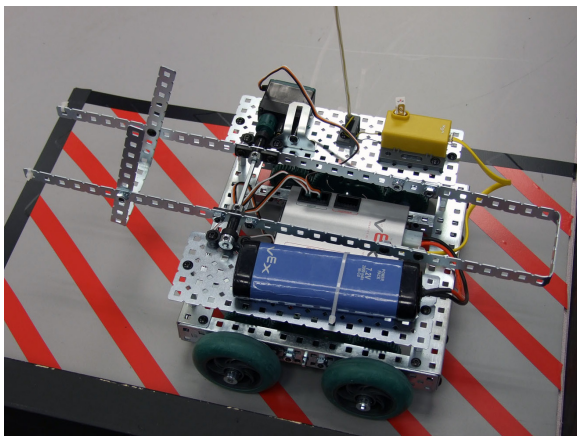
In previous programs, you used timed delays to make the robot wait a few seconds after being powered on. This gave you a few seconds to move clear of the robot before it started running the main program behaviors. This helped to improve the safety and reliability of the robot, as there was much less chance of it being thrown off course by accidental contact with the human operator who turned it on.



Thrown off course
This robot ran immediately when it was turned on, and was thrown off course as it brushed against the human operator's hand.

We fixed this problem in previous lessons by adding a timed delay between the robot being turned on and starting its run.

For the Mine Removal competition, using the timed delay allows you to start the robot before the match without fear of it *accidentally moving* during the delay period, thus avoiding both the early-start disqualification and the late-start touching penalty.



Mine Removal
One of the rules for this board is that the robot must wait for the starting signal before moving. You could be disqualified if your robot starts moving too early!

A mandatory wait period helps meet this requirement.

The downside of this approach is that it cuts into your robot's running time. The robot could potentially remain unresponsive for several seconds after the match has begun (depending on the exact timing of the person who turned the robot on).

© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems

Buttons Remote Start *(cont.)*

A better strategy for starting the robot would be to have the robot wait for a specific signal from the Radio Transmitter before running. This **remote start** command would avoid the possibility of early movement because the robot would not be processing movement commands yet. It would also avoid a touching penalty because the enabling command would be sent from the Transmitter!

```
1  task main()
2  {
3      wait1Msec(2000);
4      bIfiAutonomousMode = false;
5      bMotorReflected[port2] = 1;
6
7      ClearTimer(T1);
8      while(time10[T1] < 12000)
9      {
10       motor[port3] = vexRT[Ch3];
11       motor[port2] = vexRT[Ch2];
12     }
13
14     motor[port3] = 0;
15     motor[port2] = 0;
16 }
```

2 second delay
The current program uses a **`wait1Msec`** command to place a 2 second delay at the beginning of the program. This gives the human operator time to get out of the way before the robot starts moving.

Radio control loop
The commands in this loop set and repeatedly update the motor powers to match the values provided by the Transmitter.

Idle Loops

Sometimes, you need to simply wait for something to happen in a program. This waiting period may not have a defined length of time. Instead, you may be waiting for an event to occur, like a sensor or Radio Control value matching a certain reading. A while() loop may be a good choice for this behavior.

Consider the code to the right. It repeats an empty {body} of commands over and over until the (condition) becomes false; the effect is that the robot will do nothing until the loop ends. A loop with an empty {body} is said to idle.

```
while (vexRT[ch5] == 0)
{
}
```

(condition)
As with all while loops, the (condition) determines how long the loop continues repeating. Here, it will continue running as long as the value on Transmitter channel 5 is 0.

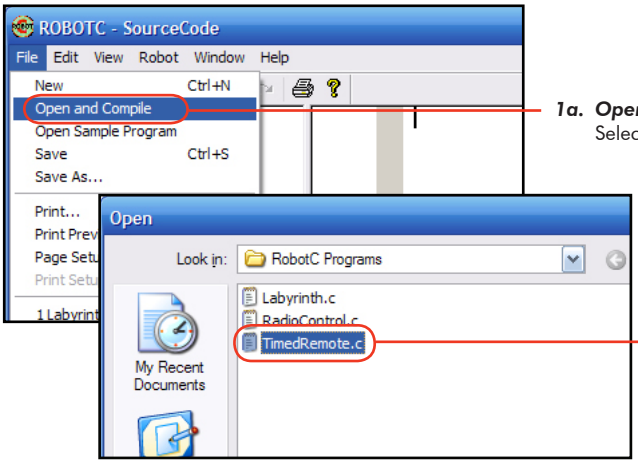
Empty {body}
There are no commands inside the loop's {body}. The loop does nothing while it continues to loop.

The "idle" looping will continue until the (condition) becomes false.

Buttons Remote Start *(cont.)*

The goal of this lesson is to replace the 2-second **`wait1Msec`** command with an **idle loop** (see previous page) that waits for a Transmitter button to be pushed before continuing on with the program.

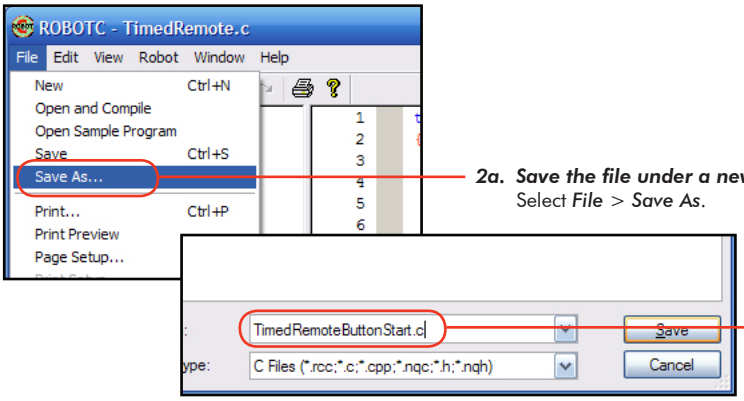
1. Open the "TimedRemote" program from the "Timers" section of this unit.



1a. Open the file
Select *File* > *Open and Compile*

1b. Open the TimedRemote program
Find *TimedRemote* in the directory where you normally save your programs, and double-click to open it (or select it and click *Open*).

2. Save the program under the new name "TimedRemoteButtonStart".



2a. Save the file under a new name
Select *File* > *Save As*.

2b. Save your program as "TimedRemoteButtonStart"
Save your program in the usual folder, with the name *TimedRemoteButtonStart*.

Buttons Remote Start *(cont.)*

3. Instead of a 2-second delay, have the robot begin the program by waiting for one of the buttons on the left-hand side of the Transmitter to be pressed.

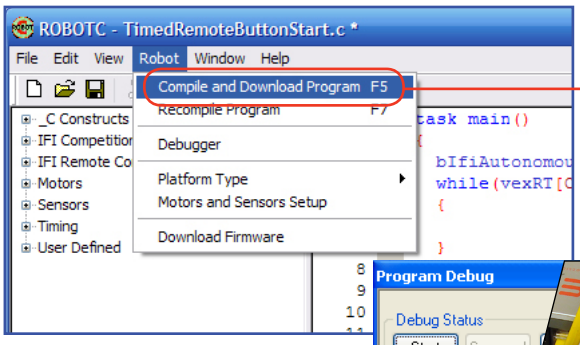
```
1  task main()
2  {
3      while(vexRT[ch5] == 0)
4      {
5
6      }
7
8      bIfiAutonomousMode = false;
9      bMotorReflected[port2] = 1;
10
11     ClearTimer(T1);
```

3a. Modify this code
Replace the **`wait1Msec`** command with an idle loop that waits for a button press on Radio channel 5.

```
1  task main()
2  {
3      bIfiAutonomousMode = false;
4      while(vexRT[ch5] == 0)
5      {
6
7      }
8
9      bMotorReflected[port2] = 1;
10
11     ClearTimer(T1);
```

3b. Move this code
You must enable radio control so you can check whether the button is pressed. Move this line to the beginning of the program.

4. Download and run your program.

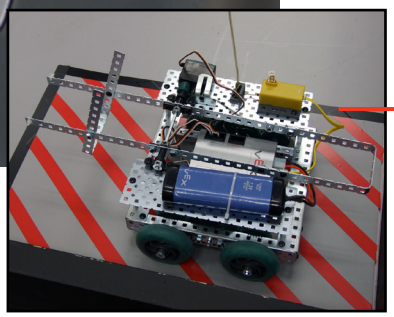


4a. Compile and Download
Make sure your robot is on and that it is plugged in with the USB cable. Then, choose *Robot* > *Compile and Download Program*.

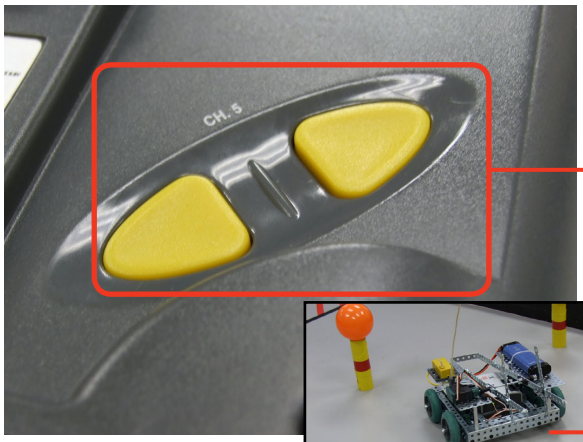
4b. Run the Program

Buttons Remote Start (cont.)

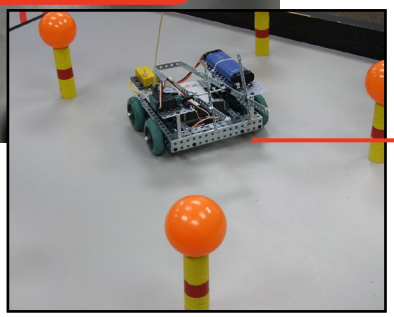
4c. Turn Transmitter ON



4d. Optional
Unplug the robot and place it on the board.



4e. Press one of the left-hand buttons
Press one of the buttons on the left-hand side of the back of the Transmitter. This will change the value on Radio channel 5 from 0 to 127 or –127.



4f. Drive the robot
The robot should now respond to driving commands as usual.

End of Section

The default value with no buttons pressed on Radio channel 5 is 0. While `vexRT[ch5]` has a value of 0, the empty idle loop continues to wait. Pressing either of the left-hand buttons on the Transmitter *changes the channel 5 value* to something other than 0 (either 127 or –127). This makes the (condition) of the loop **false** and the loop will end. The program then moves on to the next set of commands, which you wrote earlier, a second loop that enables remote control for a fixed period of time.

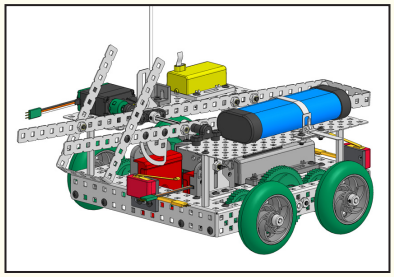
© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems

Buttons Controlling the Arm

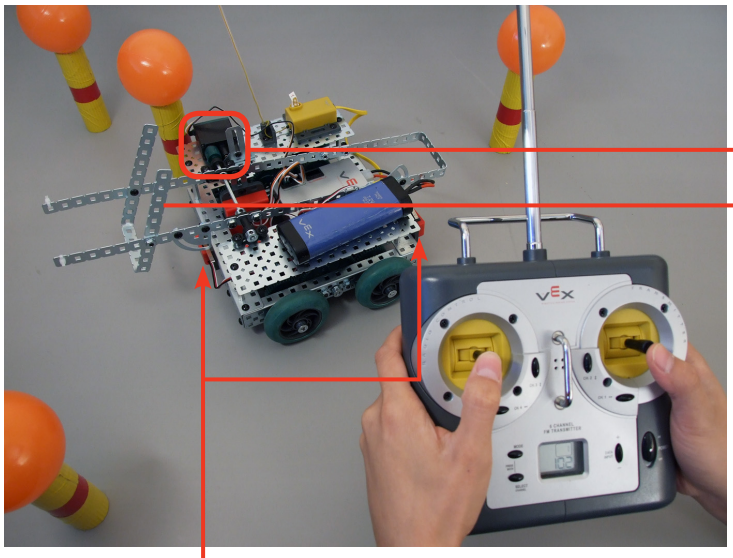
In this lesson, you will learn how to use input from the buttons on the back of the Transmitter to control the up and down movement of the robot's arm.

1. Build the manipulator arm for the robot. You may use the design shown here, or design one of your own. Note that all future references to the way the arm works and the ports it is plugged into will be based on the provided design, so you may need to adjust if your design differs significantly.

Build the Squarearm



The SquareArm design and building instructions can be found in the main menu for this lesson in Teaching ROBOTC for VEX.



Single motor design
A single motor plugged into Motor Port 6 on the VEX Micro Controller controls all movement on the arm. This motor is solely responsible for all movement of the mechanism. Running the motor in the forward direction raises the arm, while running it in the reverse direction lowers it.

Passive manipulator
The end-effector of the arm is not articulated. (In other words, the part of the arm that holds the mine cannot move to grip an object.) As a consequence, the arm relies primarily on the positioning of the main robot body to put it in the right place to accomplish its task. It also relies on the human operator to unload the mine in the disposal area.

Limit Switches
Sensors near both ends of the arm will be used to improve its performance in later lessons.

Not present on this design
Neither worm gears nor Servo Module motors are used in this design. While both of these could result in better performance, the following lessons will investigate ways to solve the same problems using software and sensors instead.

Arm Motor

motor[port6] = 127; Raises Arm
motor[port6] = -127; Lowers Arm

© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems

Buttons Controlling the Arm *(cont.)*

Checkpoint

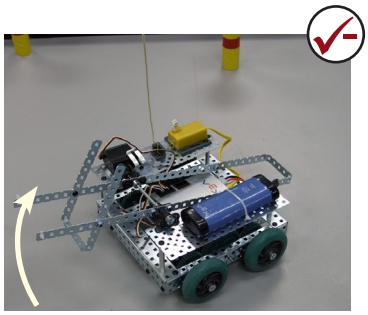
Now that you have a physical mechanism to control, let us begin on the programming side by examining whether our current programming tools are sufficient to accomplish what we need to do.

Approach #1: Direct Assignment

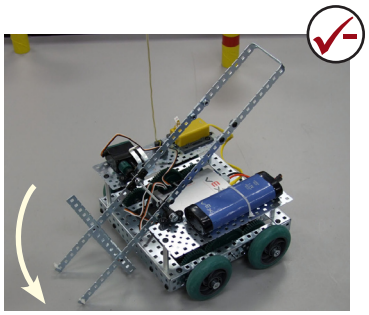
Suppose we were to simply add commands to the regular Radio control loop to run the additional motor like the others (but using the button channel instead of a joystick channel).

```
1  task main()
2  {
3      bIfiAutonomousMode = false;
4      while(vexRT[Ch5] == 0)
5      {
6
7      }
8
9      bMotorReflected[port2] = 1;
10
11     ClearTimer(T1);
12     while(time10[T1] < 12000)
13     {
14       motor[port3] = vexRT[Ch3];
15       motor[port2] = vexRT[Ch2];
16       motor[port6] = vexRT[Ch6];
17     }
18
19     motor[port3] = 0;
20     motor[port2] = 0;
21 }
```

Approach #1
Adding a direct motor power command to make the motor power run with the value (–127, 0, or 127) that the button channel provides.



Raise arm (motor power 127)
The upper button on channel 6 raises the arm while pressed, but far too quickly.



Lower arm (motor power –127)
The lower button on channel 6 lowers the arm while pressed, but also too quickly.

Maintain position (motor power 0)
Releasing the buttons maintains the position of the arm.

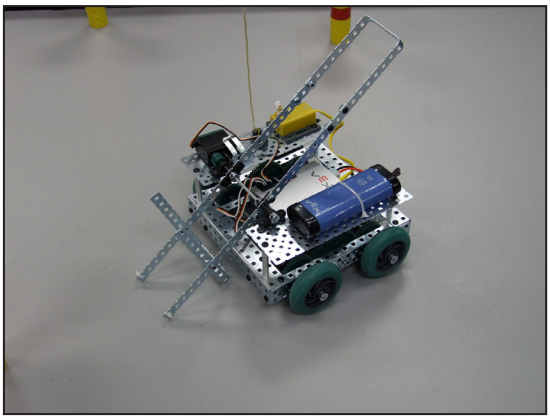
© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems

Buttons Controlling the Arm *(cont.)*

This approach works to a very limited extent. The arm can hold its position, but the commands to raise and lower the arm do so at *inappropriate power levels*.

Approach #2: Use the Signal Value to Choose an Action (using a while loop)

The previous approach ran into trouble because the *only motor powers* available to it were 127, 0, and –127. In particular, –127 and 127 were much too large to be appropriate power levels for this task.



Instead of trying to turn raw Transmitter values into motor powers, it makes more sense to have the robot instead choose motor powers based on which buttons are pressed. In the same way that you had the robot wait for the initial button press to start running, waiting for particular button presses will let you assign commands to the *top (signal value of 127), bottom (–127), and unpressed (0)* button signals.

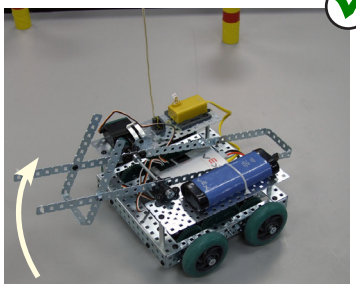
This approach uses the values from the Transmitter as a way of distinguishing between different **operator inputs**. The previous approach took the –127, 0, and 127 values as raw amounts to set the motors to. This approach uses them only as a way of distinguishing which buttons the operator is holding down, and thus as a way of signaling the desired behavior. A **`vexRT[ch6]`** signal of 127, then, would indicate that the operator is pressing the top button and would like the robot to *perform a behavior* to raise the arm (not necessarily to run the motor at exactly 127 power).

```
17      while(vexRT[Ch6] == 127)
18      {
19         motor[port6] = 31;
20      }
21      while(vexRT[Ch6] == -127)
22      {
23         motor[port6] = -31;
24      }
25      while(vexRT[Ch6] == 0)
26      {
27         motor[port6] = 0;
28      }
```

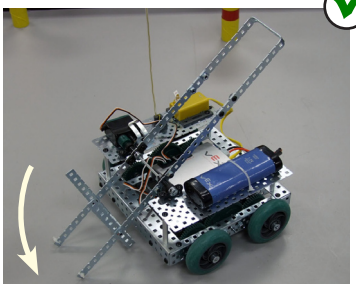
Approach #2
Using a **`while()`** loop to make the arm run at specific motor powers depending on which buttons are pushed. Transmitter values are interpreted to see which buttons are pressed, but the values are not fed "raw" into the motor powers.

Buttons Controlling the Arm *(cont.)*

Approach #2: Results

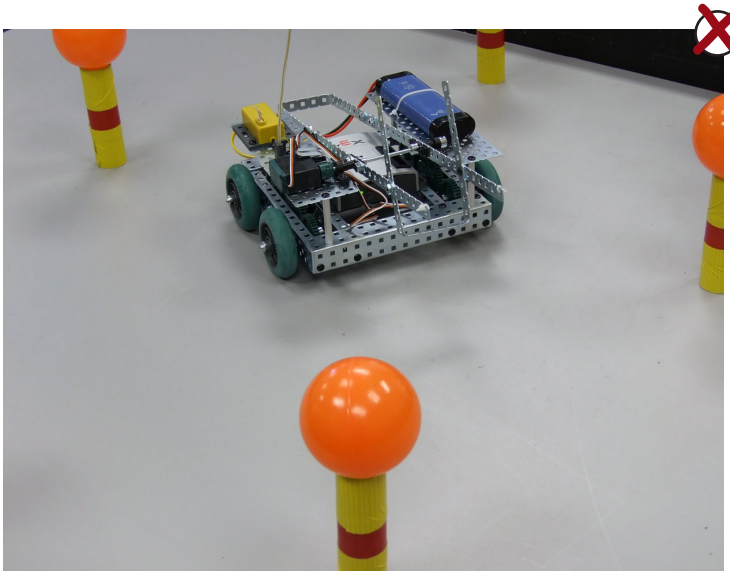


Raise arm (motor power 31)
The upper button on channel 6 raises the arm at a reasonable speed while pressed.



Lower arm (motor power -31)
The lower button on channel 6 lowers the arm at a reasonable speed while pressed.

Maintain position (motor power 0)
Releasing both buttons (or pressing both together) causes the arm to correctly maintain position, for the most part.



Drive while moving arm
The robot is not able to drive correctly while handling arm controls.

Each time an arm control is issued, the driving controls will update momentarily and then "stick" until the arm controls are changed again.

Even releasing the buttons does not solve the problem.

This approach works for controlling the arm, but does not allow the robot to drive properly.

The same property of `while()` loops that let us create the **idle loop** as a way to wait for a button press is now working against us. As the program runs through its code, it will become "trapped" inside the while loop that belongs to the current arm command. Until the Transmitter value in the (condition) changes to allow the loop to end, there is no way for the program to move on to run the other commands. This includes the commands that *update the driving motors*! As a consequence, the last driving commands that were issued to the robot remain in effect until the program can get through the blockade of `while()` loops to update the driving motors again.

© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems

Buttons Controlling the Arm *(cont.)*

Approach #3: Use the Signal Value to Choose an Action (using an `if` statement)

The portion of approach #2 that involved interpreting operator commands (rather than passing raw values) seemed to work, as the arm was functional and well-behaved by itself. The problem with the approach was in the **`while()`** loops, which kept trapping the program until the arm control values were changed on the Transmitter.

Let's investigate a replacement for the while loop in this situation. This replacement code is not a loop and therefore will not run the risk of getting stuck.

The **`if` Conditional Statement** works similarly to a **`while()`** loop. Whereas a **`while()`** loop runs {body} commands over and over again while a (condition) remains true, an **`if`** statement runs a {body} of code once if the (condition) is true, and skips past it altogether if the (condition) is false.

There is no looping involved with the **`if`** statement, so there is no risk of getting the program stuck. The {body} either runs or does not run, but in both cases the program continues on afterward.

```
if(condition)
{
  true-commands;
}
```

General form

If statements always follow the pattern shown here. If the (condition) is true, the true-commands will run. Otherwise, the program will move on instead.

Note, however, that the commands are only run *once*, and not looped!

```
14          motor[port3]    vexRT[Ch3];
15          motor[port2] = vexRT[Ch2];
16
17          if(vexRT[Ch6] == 127)
18          {
19             motor[port6] = 31;
20          }
21          if(vexRT[Ch6] == -127)
22          {
23             motor[port6] = -31;
24          }
25          if(vexRT[Ch6] == 0)
26          {
27             motor[port6] = 0;
28          }
29       }
```

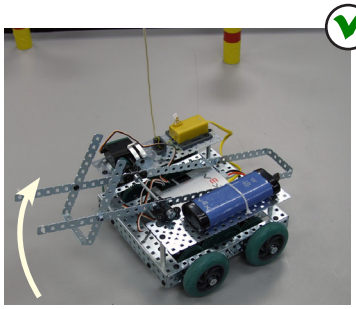
Approach #3

Using an **`if`** statement to set the arm to run at specific motor powers depending on which buttons are pushed. The **`if`**-statement will decide whether or not to run certain {bodies} of code, but will not loop them and risk getting stuck. Transmitter values are still interpreted, rather than used raw.

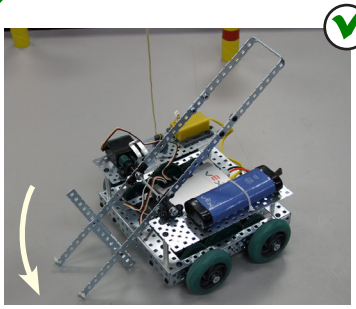
© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems

Buttons Controlling the Arm *(cont.)*

Approach #3: Results

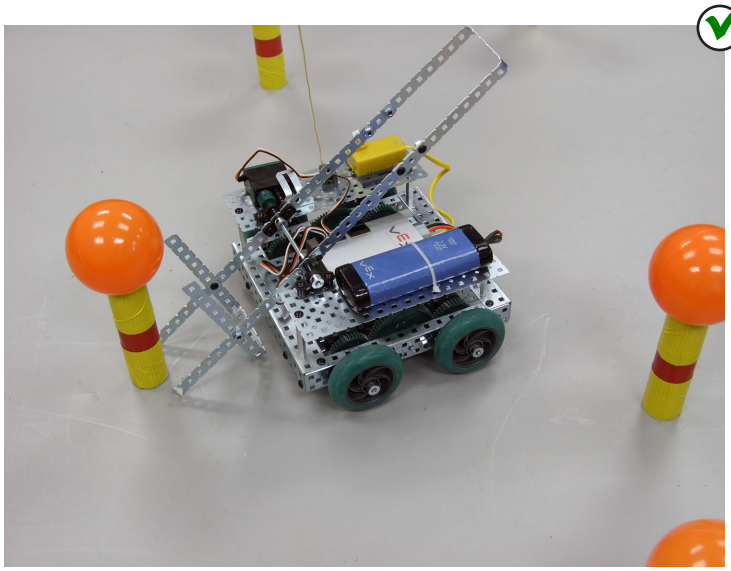


Raise arm (motor power 31)
The upper button on channel 6 correctly raises the arm while pressed.



Lower arm (motor power -31)
The lower button on channel 6 correctly lowers the arm while pressed.

Maintain position (motor power 0)
Releasing both buttons (or pressing both together) causes the arm to correctly maintain position.



Drive while moving arm
The robot is able to drive and move the arm at the same time without any interference.

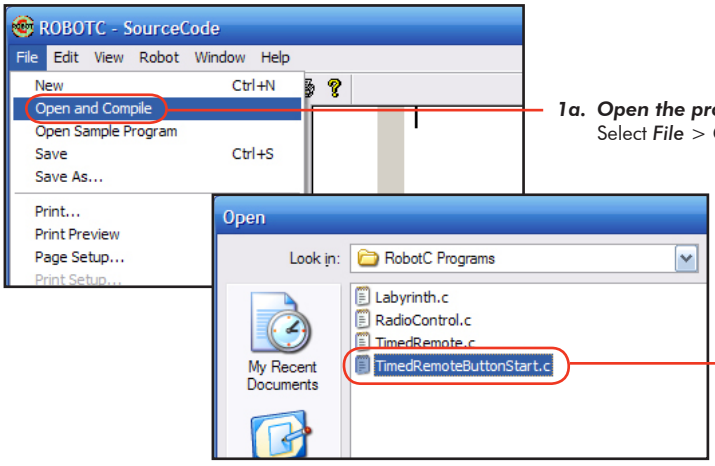
The use of the **`if`** statement inside the main program loop gives the robot the opportunity to respond to any of the Transmitter button presses, but eliminates the risk of getting stuck inside a button-checking **`while()`** loop.

We will use this **`if`** statement-based approach for the program.

© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems

Buttons Controlling the Arm *(cont.)*

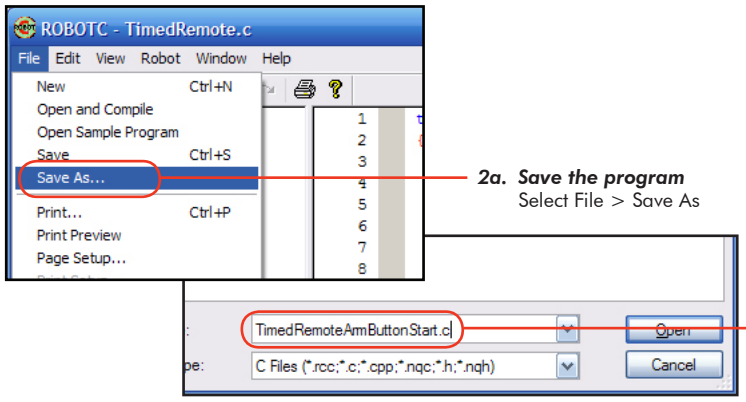
1. Start with the program from the previous section, which uses the buttons to perform a remote start.



1a. Open the program
Select *File* > *Open and Compile*

1b. Open the TimedRemoteButtonStart program
Find TimedRemoteButtonStart in the directory where you normally save your programs, and double-click to open it (or select it and click *Open*).

2. Save this program under the new name "TimedRemoteArmButtonStart".



2a. Save the program
Select File > Save As

2b. Save As "TimedRemoteArmButtonStart"
Save your program under the name *TimedRemoteArmButtonStart* in the folder where you usually save your programs.

Buttons Controlling the Arm *(cont.)*

3. Add the following lines to your main timed program loop to handle the three possible button values. Remember that because the values are being used only to determine which buttons are pushed, the actual motor powers you set can be much more reasonable than "full power."

```
1  task main()
2  {
3       bIfiAutonomousMode = false;
4       while(vexRT[Ch5] == 0)
5       {
6
7       }
8
9       bMotorReflected[port2] = 1;
10
11      ClearTimer(T1);
12      while(time10[T1] < 12000)
13      {
14         motor[port3] = vexRT[Ch3];
15         motor[port2] = vexRT[Ch2];
16
17         if(vexRT[Ch6] == 127)
18         {
19            motor[port6] = 31;
20         }
21         if(vexRT[Ch6] == -127)
22         {
23            motor[port6] = -31;
24         }
25         if(vexRT[Ch6] == 0)
26         {
27            motor[port6] = 0;
28         }
29      }
30
31      motor[port3] = 0;
32      motor[port2] = 0;
33 }
```

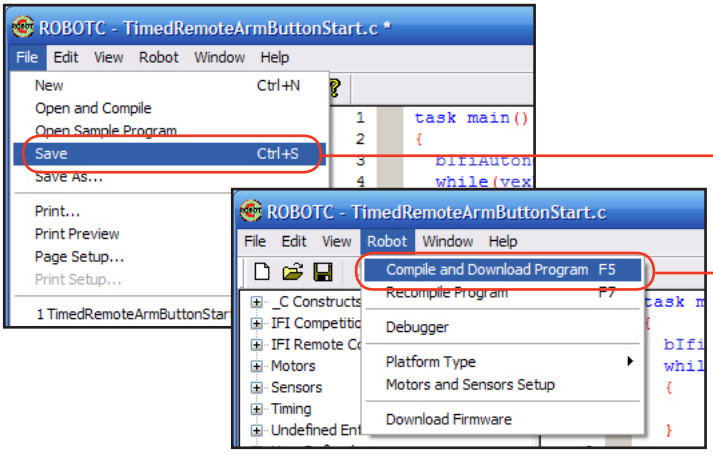
3a. Add this code
The `if` statement checks the (condition) to determine whether the {body} should run.

In this case, the (condition) will be true if the value sent on Radio channel 6 is equal to 127 (Transmitter right-hand top button is pushed). If this is true, the robot will run the arm motor (port6) at one-quarter power in the forward direction. If it is false, the program will simply ignore the {motor command} and continue on with the next line.

3b. Add this code
If the value sent on Radio channel 6 is equal to –127 (Transmitter right-hand bottom button is pushed), the robot will run the arm motor (port6) at one-quarter power in the reverse direction.

3c. Add this code
If the value sent on Radio channel 6 is equal to 0 (neither right-hand button pushed, or both pushed), the robot will hold the arm in position.

4. Save, compile, and download this program.

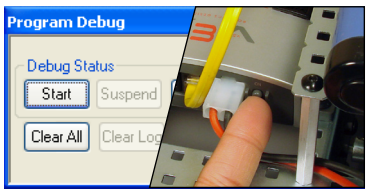


4a. Save your program
Select *File* > *Save*.

4b. Compile and Download
Make sure your robot is on and that it is plugged in with the USB cable. Then, choose *Robot* > *Compile and Download Program*.

Buttons Controlling the Arm *(cont.)*

5. Run the program by either pressing the Start button, or switching your robot off and then on.



5. Run the Program

End of Section

The arm on the robot is now functional, and can be operated in conjunction with the existing movement and steering controls. By using decision-making **control statements**, we were able to distinguish between the raw transmitter values, and decide exactly how the robot should respond to each. By using an `if` statement rather than a `while()` loop, the robot was able to check all the possible values of the Transmitter buttons without getting stuck inside the loop when it went to run the loop's behaviors.

Your robot now has the basic capabilities needed to compete in the Mine Removal challenge. These capabilities are limited, however, and your team must now work to improve the robot's design, smooth out its performance, and refine the strategies that you will use. Major gains are within reach through effective use of the mechanical and programming skills you have now. However, others can only be achieved by combining the two at a deeper level. When you are ready, move on to the next unit on Sensors.

© Carnegie Mellon Robotics Academy / For use with VEX® Robotics Systems