Pseudocode & Flow Charts

Pseudocode is a **shorthand notation for programming** which uses a combination of **informal programming structures and verbal descriptions of code.** Emphasis is placed on expressing the behavior or outcome of each portion of code rather than on strictly correct syntax (it does still need to be reasonable, though).

In general, pseudocode is used to outline a program before translating it into proper syntax. This helps in the initial planning of a program, by creating the logical framework and sequence of the code. An additional benefit is that because pseudocode does not need to use a specific syntax, it can be translated into different programming languages and is therefore somewhat universal. It captures the **logic and flow of a solution** without the bulk of strict syntax rules.

Below is some pseudocode written for a program which controls a motor and an LED as long as a touch sensor is not pressed. A motor turns on and an LED turns off if no object is deteced within 20cm of a sonar sensor; the motor turns off and an LED turns on if an object is detected within 20 cm.

```
task main()
{
 while ( touch sensor is not pressed )
  {
    if(sonar detects object > 20cm away)
    {
      Right Motor runs forward
      Red LED turns off
    }
    else
    {
      Right Motor stops
      Red LED turns on
    }
  }
}
```

Some intact syntax
The use of a while loop in the pseudocode is fitting because the way we read a while loop is very similar to the manner in which it is used in the program.

Descriptions
There are no actual motor commands in this section of the code, but the pseudocode suggests where the commands belong and what they need to accomplish.

This pseudocode example includes elements of both programming language, and the English language. Curly braces are used as a visual aid for where portions of code need to be placed when they are finally written out in full and proper syntax.

Flow Charts are a **visual representation of program flow**. A flow chart normally uses a combination of **blocks** and **arrows** to represent actions and sequence. Blocks typically represent **actions**. The **order** in which actions occur is shown using arrows that point from statement to statement. Sometimes a block will have multiple arrows coming out of it, representing a step where a **decision** must be made about which path to follow.

Start and End symbols are represented as rounded rectangles, usually containing the word "Start" or "End", but can be more specific such as "Power Robot Off" or "Stop All Motors".

Start/Stop

Actions are represented as rectangles and act as basic commands. Examples: "wait 1 second"; "increment LineCount by 1"; or "motors full ahead".

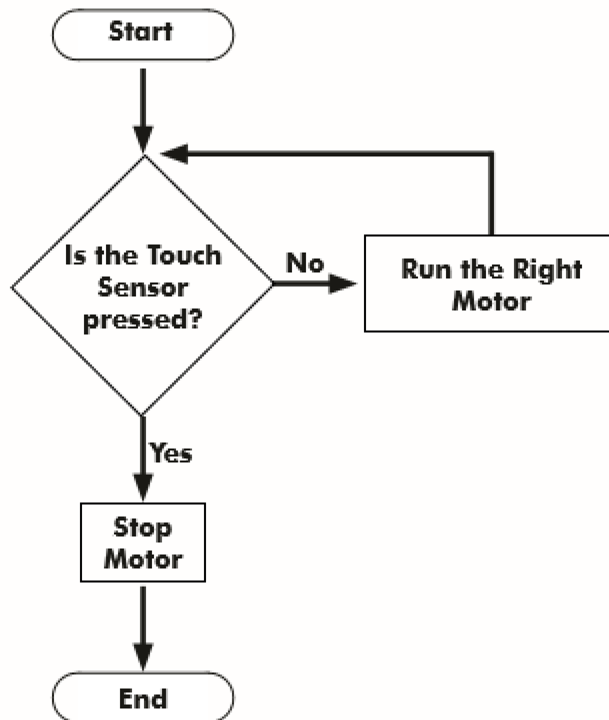
Action

Decision blocks are represented as diamonds. These typically contain Yes/No questions. Decision blocks have two or more arrows coming out of them, representing the different paths that can be followed, depending on the outcome of the decision. The arrows should always be labeled accordingly.

Decision

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

To the right is the flow chart of a program which instructs a robot to run the right motor forward as long as its touch sensor is not pressed. When the touch sensor is pressed the motor stops and the program ends.



To read the flow chart:

- Start at the "**Start**" block, and follow its arrow down to the "Decision" block.
- The **decision block** checks the status of the touch sensor against two possible outcomes: the touch sensor is either pressed or not pressed.
- If the touch sensor is not pressed, the program follows the **"No" arrow** to the action block on the right, which tells the right motor to run forward. The arrow leading out of that block points back up and around, and ends back at the Decision block. This forms a **loop**!
- The **loop** may end up repeating many times, as long as the Touch Sensor remains unpressed.
- If the touch sensor is pressed, the program follows the "Yes" arrow and stops the motors, then ends the program.

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

PROGRAMMING BASICS IN ROBOTC

VERSION CONTROL: It is a good idea to always save your file as a new filename, date changes were made, who is making the changes, and always document what changed in that version. If you do have an issue with a program, you can always go back one version and not lose all your work. Here is an example:

Filename	Date	Programmer	What changed
BEST2015.c	10/5/2015	Bob	Remote control -defined driving motors
BEST2015A.c	10/15/2015	Mary	Remote control-added the lift motors
BEST2015B.c	10/25/2015	Bob	Remote control-added the scoop motor
BEST2015C.c	11/2/2015	Bob	Remote control-added a bumper sensor to control the height of the lift.

White Space

White Space is the use of **spaces, tabs, and blank lines** to visually organize code. Programmers use White Space since it can group code into sensible, readable chunks without affecting how the code is read by a machine. For example, a program that would run a robot forward for 2 seconds, and then backward for 4 seconds, could look like either of these:

Program Without White Space

```
task main()
{
motor[motorC]=100;
motor[motorB]=100;
wait1Msec(2000);
motor[motorC]=-100;
motor[motorB]=-100;
wait1Msec(4000);
}
```

Program With White Space

```
task main()
{

  motor[motorC]=100;
  motor[motorB]=100;
  wait1Msec(2000);

  motor[motorC]=-100;
  motor[motorB]=-100;
  wait1Msec(4000);

}
```

Both programs will perform the same, however, the **second uses white space** to organize the code to **separate the program's two main behaviors**: moving forward and moving in reverse. In this case, line breaks (returns) were used to vertically segment the tasks. Horizontal white space characters like spaces and tabs are also important. Below, white space is used in the form of indentations to indicate which lines are within which control structures (task main, while loop, if-else statement).

Program Without White Space

```
task main()
{
while(true)
{
if(SensorValue(touch)==0)
{
motor[motorA]=100;
motor[motorC]=100;
}
else
{
motor[motorA]=-100;
motor[motorC]=-100;
}
}
}
```

Program With White Space

```
task main()
{
  while(true)
  {
    if(SensorValue(touch)==0)
    {
      motor[motorA]=100;
      motor[motorC]=100;
    }
    else
    {
      motor[motorA]=-100;
      motor[motorC]=-100;
    }
  }
}
```

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

Comments

Commenting a program means using descriptive text to explain portions of code. The compiler and robot both ignore comments when running the program, allowing a programmer to leave important notes in non-code format, right alongside the program code itself. This is considered very good programming style, because it cuts down on potential confusion later on when someone else (or even you) may need to read the code.

There are two ways to mark a section of text as a comment rather than normal code:

Type	Start Notation	End Notation
Single line	//	(none)
Multiple line	/*	*/

Below is an example of a program with single and multi-line comments. Commented text turns green.

```
/*
   This program uses commenting
   to describe each process.
*/
task main()
{
  bMotorReflected[port2] = 1; //reflect rotation on port 2
  motor[port3]=127;    //port3 receives full power
  motor[port2]=127;    //port2 receives full power
  wait1Msec(5000);     //both motors run for 5 sec.
}
```

"Commenting out" Code

Commenting is also sometimes used to temporarily "disable" code in a program without deleting it. In the program below, the programmer has code to run straight and then turn right. However, in order to test just the first part of the program, the programmer made the second behavior into a comment, so that the robot would ignore it. When the programmer is done testing the first behavior, he/she will remove the // comment marks to re-enable the second behavior in the program.

```
task main()
{
  bMotorReflected[port2] = 1;
  motor[port3]=127;
  motor[port2]=127;
  wait1Msec(5000);

  //motor[port3]=127;
  //motor[port2]=-127;
  //wait1Msec(1500);
}
```

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

Boolean Logic

ROBOTC **control structures** that make decisions about which pieces of code to run, such as **while loops** and **if-else** conditional statements, always depend on a (condition) to make their decisions. **ROBOTC (conditions) are always Boolean statements.** They are always either true or false at any given moment. Try asking yourself the same question the robot does – for example, whether the value of the Ultrasonic Sensor is greater than 45 or not. Pick any number you want for the Ultrasoni Sensor value. The statement "the Ultrasonic Sensor's value is greater than 45" will still either be true or be false.

Condition	Ask yourself...	Truth value
`SensorValue(sonarSensor) > 45`	Is the value of the Ultrasonic Sensor greater than 45?	**True**, if the current value is more than 45 (for example, if it is 50). **False**, if the current value is not more than 45 (for example, if it is 40).

Some (conditions) have the additional benefit of ALWAYS being true, or ALWAYS being false. These are used to implement some special things like "infinite" loops that will never end (because the condition to make them end can never be reached!).

Condition	Ask yourself...	Truth value
`1==1`	Is 1 equal to 1?	**True**, always
`0==1`	Is 0 equal to 1?	**False**, always

Comparison Operators

Comparisons (such as the comparison of the Ultrasonic sensor's value against the number 45) are at the core of the decision-making process. A well-formed comparison typically uses one of a very specific set of operators, the "comparison operations" which generate a true or false result. Here are some of the most common ones recognized by ROBOTC.

ROBOTC Symbol	Meaning	Sample comparison	Result
==	"is equal to"	`50 == 50`	**true**
		`50 == 100`	**false**
		`100 == 50`	**false**
!=	"is not equal to"	`50 != 50`	**false**
		`50 != 100`	**true**
		`100 != 50`	**true**
<	"is less than"	`50 < 50`	**false**
		`50 < 100`	**true**
		`100 < 50`	**false**
<=	"is less than or equal to"	`50 <= 50`	**true**
		`50 <= 100`	**true**
		`50 <= 0`	**false**
>	"is greater than"	`50 > 50`	**false**
		`50 > 100`	**false**
		`100 > 50`	**true**
>=	Greater than or equal to	`50 >= 50`	**true**
		`50 >= 100`	**false**
		`100 >= 50`	**true**

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

Use in Control Structures

"Under the hood" of all the major decision-making control structures is a simple check for the Boolean value of the (condition). The line `if (SensorValue(bumper) == 1)...` may read easily as "if the bumper switch is pressed, do...", but the robot is really looking for `if(true)` or `if(false)`. Whether the robot runs the "if true" part of the if-else structure or the "else" part, depends solely on whether the (condition) boils down to `true` or `false`.

```
if (50 > 45) ...
if (true) ...
```

Logical Operators

Some (conditions) need to take **more than one thing** into account. Maybe you only want the robot to run if the traffic light is green AND there's no truck stopped in front of it waiting to turn. Unlike the comparison operators, which produce a truth value by comparing other types of values (is one number equal to another?), the **logical operators** are used to **combine multiple truth values into one single truth value.** The combined result can then be used as the (condition).

Example:
Suppose the value of a Light Sensor named **sonarSensor is 50**, and at the same time, the value of a Bumper Switch named **bumper is 1** (pressed).

The Boolean statement `(sonarSensor > 45) && (bumper == 1)` would be evaluated...v

```
(50 > 45) && (1 == 1)
true && true
```

ROBOTC Symbol	Meaning	Sample comparison	Result
&&	"AND"	`true && true`	**true**
		`true && false`	**false**
		`false && true`	**false**
		`false && false`	**false**
||	"OR"	`true || true`	**true**
		`true || false`	**true**
		`false || true`	**true**
		`false || false`	**false**

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

While Loop

A while loop is a structure within ROBOTC which allows a portion of code to be run over and over, as long as a certain condition remains true.

Below is the pseudocode outline of a while loop.

```
while(condition)
{
  // repeated-commands
}
```

(condition)
Either **true** or **false** (see Reference > Boolean Logic).

Repeated commands
Commands placed here will run over and over as long as the (condition) is **true** when the program checks at the beginning of each pass through the loop.

Below is an example of a program using a While Loop.

```
task main()
{
 while(nMotorEncoder[motorC]<360)
 {
   motor[motorC]=100;
   motor[motorB]=100;
 }
}
```

The condition is true as long as the rotation sensor detects less than 360 degrees of rotation.

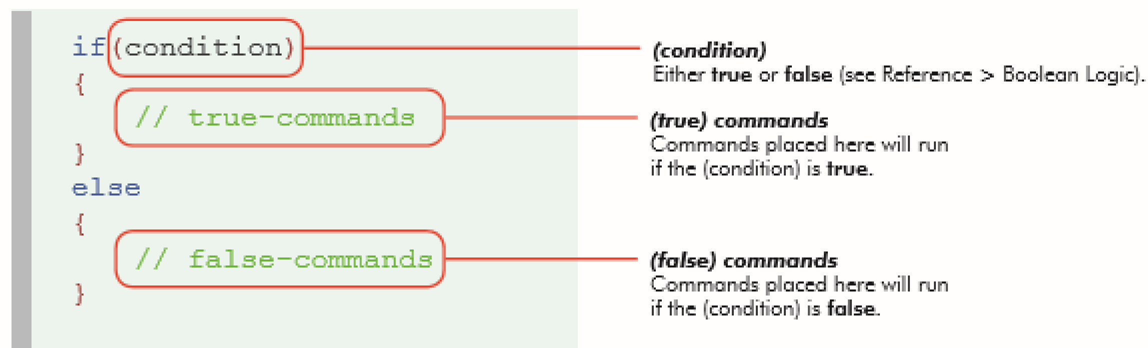
While the condition is true, both motors will receive 100% power.

This while loop runs as long as the rotation sensor detects less than 360° of spin. As long the condition remains true, motor A and motor B are told to run at full power. When the condition becomes false, i.e. the rotation sensor detects more than 360° of spin, the loop ends and the robot goes on to the next line of code. In this case, the program ends.

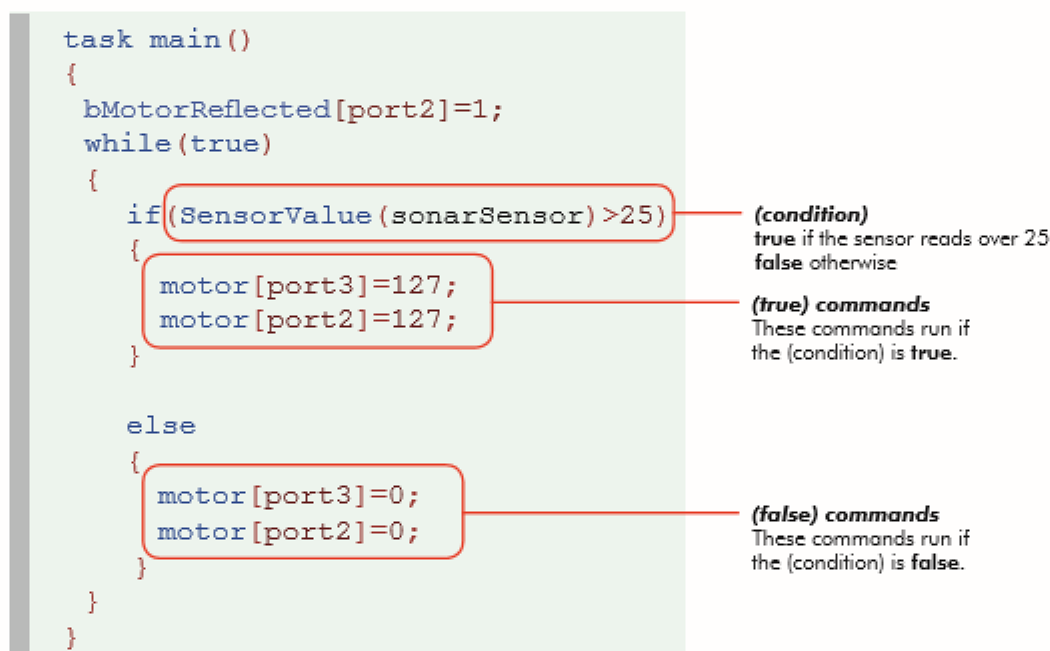
if-else Statement

An if-else Statement is one way you allow a computer to make a decision. With this command, the program will check the (condition) and then execute one of two pieces of code, depending on whether the (condition) is true or false.

Below is the pseudocode outline of an if-else Statment.



Below is an example program containing an if-else Statement.



This if-else Statement tells the robot to run both motors at full power if the nearest object the Ultrasonic Range Finder detects is more than 25 inches away. If the Ultrasonic Range Finder detects an object closer than 25 inches, then the "else" portion of the code will be run and the robot will stop moving. The outer `while(true)` loop makes the if-else statement run over and over forever.

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