



K-Botics, Team 2809

Tom Radcliffe, Ph.D., P.Eng
2011

www.kbotics.ca

Representing Robots

This note is a collection of practical tips on robotic programming. A lot of them apply to C++ programming in general.

The unifying idea behind most of these suggestions is that programs should be as close in structure to the hardware system it represents as possible. This is sometimes called the Principle of Verisimilitude.

It is worth remembering that human beings are very bad at two things: thinking, and communicating. Keeping our code structure as close as possible to the structure of the machine it represents makes it easier to think about and talk about.

The FRC C++ framework provides a number of base classes for representing robots. This note will talk about things in terms of the `IterativeRobot` class, but the principles generalize beyond that.

Declarations and Organization

C++ classes typically have a separate header file where the class members are declared. Declarations should be grouped logically by type and/or sub-system.

For example, a robot with both drive motors and motors on an arm or manipulator, might have two blocks of declarations for motors: one for the arm and one for the drive motors.

Likewise, inputs and outputs of similar type ought to be grouped together: `DigitalInput` objects, `Solenoid` objects, and so on. If there are few enough of each of these that they can be taken in at a glance (typically fewer than ten) then they can be put in any order. If there are more than five or ten it is worth maintaining them in alphabetical order.

Keeping to simple rules of this kind makes it much, much easier to find various parts of the code. Finding things easily can become remarkably important in the later phases of development, when the code has had time to accumulate a lot of junk.

The FRC Robot Lifecycle

`IterativeRobot` has four major lifecycle phases:

- 1) Construction and `RobotInit`
- 2) Disabled mode
- 3) Autonomous mode
- 4) Teleoperated mode

The following sections will deal with each of these in turn. The various modes all have an initialization and periodic method associated with them, which will be dealt with in turn.



K-Botics, Team 2809

Tom Radcliffe, Ph.D., P.Eng
2011

www.kbotics.ca

Representing Robots

Construction and Naming

Representation begins with the constructor: this is where we build all the software peices that represent the robot's hardware. Naming is an important aspect of representation: we want to be able to use the same language to talk about the software that we do to talk about the hardware.

Don't be afraid to be wordy in naming. For example, if you have a limit switch that is triggered by your robot's left handed widget, call it something like `m_LeftHandedWidgetLimit`.

I generally favour a very simplified form of "Hungarian Notation", prefixing floating point variable names with "f", integer types with "n" and pointer types with "p". Member variables should be prefixed with "m_" to clearly distinguish them from local variables. So an integer member variable that counts widget turns might be named "`m_nWidgetTurnCount`". I also favour CamelCase for ease of readability.

The recommendations of the preceding paragraph are just personal preferences, and preferences differ. However, it is important that there be some agreement on what conventions to use. There are many good coding standards available on the Web, and it's worth putting some time into reviewing them and adopting one. Code that is written to a common standard is easier to read: it communicates more clearly and requires less thinking, both of which are good things.

Disabled Mode

The cRIO file system is available via the usual C++ streams, and anything written to the console can be captured on the NI console during debugging. This can be very useful in disabled mode for testing your robot.

In `DisabledPeriodic` we typically add a counter the lets us output some subset of sensors every second or so. This makes testing and verification of the robot on the bench very easy. For example:

```
void MyRobot::DisabledPeriodic()
{
    static int nCounter = 0;

    if (50 == nCounter) // assuming loop is running at 50 Hz
    {
        std::cerr << m_pSomeSensor->GetValue() << std::endl;
        nCounter = 0;
    }
    ++nCounter;
}
```

This can be useful for testing algorithms as well as sensor inputs.



K-Botics, Team 2809

Tom Radcliffe, Ph.D., P.Eng
2011

www.kbotics.ca

Representing Robots

Autonomous Mode

AutonomousInit is called before autonomous mode starts, and should be used to ensure that robot is in a known state. For some sensors, like the gyro, this means resetting them to a zero state.

In both autonomous and teleop modes maintaining state is a significant problem. The robot needs to remember what it's doing, and this can get complicated very fast. The most dramatic source of complexity is the need to deal with hardware that isn't quite performing up to spec, or which may have broken down entirely, or which may never have quite worked properly in the first place. Part of the fun of software development for an FRC build is that the robot is a moving target, and this is a source of enormous complexity in the code.

In autonomous mode in particular you need to think about various ways of carrying out the mission with limited operational hardware. There's usually a simple approach that doesn't rely on a lot of sensor inputs, but only accomplishes the most basic functions. Then there will be more complicated ways depending on what sensors and actuators are working properly.

In general it is a good idea to have a fallback method in the code that will work when not much else is. In the case of the 2011 Logomotion game this might mean some simple pre-programmed pattern that depends only on the robot being placed correctly at the start.

Then there can be one or two more complex methods that depend on more things working: line sensors and so on. Just keep in mind the reality that wires get pinched and broken, sensors sometimes fail spontaneously or get knocked loose of their mountings, and in general failures happen that you can't possibly imagine. What we can or cannot imagine does not determine what can or cannot happen, and in competition sometimes the most unlikely things occur. On that basis, simple and robust is to be preferred over complex and delicate.

Given that you won't know the state of the robot until it leaves the pit, how best to decide what method to use? A simple switch feeding into a digital input can be enough to do the job: just test the switch state in AutonomousInit and set a state variable that will be used to decide which method to run with. The only issue is that the switch itself might not work, or the toggle may get broken off, making the switch inoperable.

One of the most remarkable things about human sensory systems is that they are full of internal consistency checks that tell us when they are transducing reality correctly. When these checks fail we experience a sensory illusion, which tells us there is something weird going on. If our senses didn't work almost perfectly most of the time we would never be aware of illusions because weird experiences would be an ordinary part of life. As it is, illusions are rare and exceptional, telling us that most of the time our senses react to the world pretty consistently.

A robot's sensorium is a lot more limited, and doesn't provide much self-checking ability, so you will often have to simply trust that there aren't any evil demons interfering with the sensors, rather than looking for internal consistency the way humans can.



K-Botics, Team 2809

Tom Radcliffe, Ph.D., P.Eng
2011

www.kbotics.ca

Representing Robots

Teleoperated Mode

The program needs to do one thing: make the driver's life as easy as possible.

What this means depends on the game, and the driver. Different drivers have different preferences, and it's always a good idea to field test and update your control code to work well with your driver.

Some games have a lot of pushing and shoving, and driver control is vital. In other cases there is room for automated push-button aids that let the robot perform simple tasks with minimal driver intervention.

One area where the computer can really help out is in tracking a straight line. Any real drive-train will tend to have slightly different forces on each side, resulting in a tendency to turn even while the driver tells the robot to move in a straight line. By adding a gyro sensor and setting it to zero whenever the robot is being turned by the driver, you can then use a simple differential or PID controller to keep the robot on track when driving straight. Our code for this looks something like this:

```
void MyRobot::TelopPeriodic()
{
    float fX = m_pStick->GetX();
    float fY = m_pStick->GetY();

    float fR = m_pGyro->GetValue();

    if (fabs(fX) < 0.1) // not much side-stick
    {
        fX = 0.2*(fR-m_fGyroSetPoint); // adjust factor and sign to suit
    }
    else // driver is turning robot, update gyro setpoint
    {
        m_fGyroSetPoint = fR;
    }

    ...use fX, fY to set wheel speeds...
}
```

The member variable `m_fGyroSetPoint` should be initialized to zero in `TeleopInit()` to ensure there are no surprises on the first call to `TeleopPeriodic`. Also, if you get the sign wrong in the gyro correction the robot will spin out of control, so make sure you test things with the robot off the ground, and keep the stop button handy.

There are a wide variety of tasks like this: line following, for example. The driver should have a simple way of signaling to the robot that it should follow a line. This may be as simple as doing so by default unless the driver tells it otherwise, or it may involve holding a button down or toggling a button state. Find out what works for your driver and see.



K-Botics, Team 2809

Tom Radcliffe, Ph.D., P.Eng
2011

www.kbotics.ca

Representing Robots

Conclusion

This note has touched on a few practical tips for programming your robot. There's a lot more that hasn't been said. Familiarity with the tools--the WindRiver Eclipse-based development environment--and the C++ language is important. A good introductory C++ book is hard to find, but the Effective C++ and Effective STL books are full of practical advice that will be useful to anyone who knows the basics of the language. A good Eclipse book is impossible to find, so there's really no substitute for hands-on experience with WindRiver. For non-robotics projects Wascana Eclipse provides a decent C++ development environment in Eclipse (not comparable to Microsoft Visual Studio, but using that won't build your Eclipse skills!)