Asynchronous Programming with Qt

Paul R. Potts

October 2019

I recently became bogged down for a couple of days by problems with an Windows application, written in C++, using the Qt framework. I needed to get a pretty simple thing working. The Windows application sends packets of data to the microcontroller over a couple of different possible interfaces. You can plug in a USB cable, and send the data using a library call to the driver for the chip on the other end, which converts from USB on one side to old-fashioned serial on the other. Or if you have an adapter from USB to RS-232 or an older computer with an honest-to-God built-in RS-232 port, you can use that.

And so, it's time to talk about *asynchronicity*.

Asynchronicity

The USB connection was working perfectly. I could reliably send firmware updates to the microcontroller that way. But the serial side, which uses a C++ class called, unsurprisingly, **QSerialPort**, was not working reliably at all. So I had some debugging to do.

Sending data with this class is pretty straightforward. You just call one method to send a chunk of data. The call doesn't *block* — that is, the method call doesn't wait until all the data has gone out before returning. Computers run *much* faster than serial ports do, so it doesn't really make sense to stop your whole program, or even one thread of it, while the data is being slowly clocked out onto the wire.

That makes things a bit tricky, though, because I actually want my code to stick with a rigid "call-and-response" plan. I want to send the packet, and wait for the answer before I send the next packet. But the call to receive data *also* does not "block," at least not in the usual sense, and again, you don't usually want it to, since it brings your thread of execution to a halt. Depending on how your program is structured, that might also mean that the graphical user interface would grind to a halt, too, and that's bad.

So there are two approaches described in the available sample code, synchronous (blocking) and asynchronous (non-blocking). "Asynchronous" means "not synchronized" with your code; it means that the work happens behind the scenes,

and it will be done at some point, and so you have to be ready to handle some kind of notification that tells you when it is done.

Qt is a very old framework, and it came into existence before modern versions of C++ existed, and when computers were much slower, and it wasn't so commonplace for applications with graphical user interfaces to have multiple threads of execution. So the QSerialPort class's way of supporting asynchronous sending and receiving is kind of weird, and not very much like the way most modern libraries do it.

A more modern library would, typically, let me call a function to send data, with parameters that included a pointer to a buffer of data to send, a count of the number of bytes in the buffer, and a *timeout* value, typically in milliseconds. When you make a call like this, you're telling the operating system or library or whatever "hey, here's some data; send it out the serial port. Wait for up to 25 milliseconds. If it takes longer than that, stop and let me know." Then that call would return some sort of error code letting you know whether is succeeded or not. This call would "block," but you wouldn't typically care, because your application would be broken into threads, so that if one thread blocked, the others would keep running, and parts like the graphical user interface would not freeze up. My application is already broken into threads; this code is running in a thread to do serial communication by either RS-232 or USB, and it is separate from the thread that runs the graphical user interface.

Anyway, that's one way to send data. It's synchronous, but that doesn't really matter, because only one thread waits for the data to go out. Receiving can be done in a similar way: provide the number of bytes you expect to get, a pointer to a buffer big enough to hold those bytes, and a timeout. One thread waits for data to come in, but the others keep going.

The relevant part of Qt's synchronous serial receive sample code looks like this:

```
if (serial.waitForReadyRead(currentWaitTimeout)) {
    QByteArray responseData = serial.readAll();
    while (serial.waitForReadyRead(10))
        responseData += serial.readAll();

    const QString response = QString::fromUtf8(responseData);
    emit this->response(response);
} else {
    emit timeout(tr("Wait read response timeout %1")
                 .arg(QTime::currentTime().toString()));
}
```

If you squint, you can kind of see how that looks like the process I described, except that you can't specify how many bytes you want. In fact this example doesn't guarantee you'll get any particular number of characters. That's fine if you have an ongoing stream of characters that might trickle in at any time, and you just want to log them, or something like that, and then stop if you haven't

gotten any for a while. But that's not at all how my application works. I need to get my packet with its fixed number of bytes, because the contents of the packet will tell my code what to do next.

Typically, libraries also provide a way to read and write data asynchronously. This is typically done using some kind of *callback* — the library will "call back" to a function in your code, and that function needs a way of signaling to another part of your code that the data was sent, or received, or that it didn't work.

In that case, you might want the code in your class object to wait on a *semaphore*; that's the classic "computer science-y" thing to do. Your code waits for a "resource" to be acquired. A semaphore is a flexible mechanism that can be used to manage any kind of resource. In this case of reading data, the "resource" is the packet of data we are waiting for. The called-back function "gives" the semaphore, and my main thread of execution tries to "take" the semaphore. There's a timeout. This is a classic consumer/producer problem. Qt provides a class called **QSemaphore** that seems like it ought to be perfect for this. So I implemented some code kind of like the example below. (This is not the real code; it's a simplified excerpt to illustrate the concept).

First, we need a method that gets called when bytes are ready to read. This method then receives all the bytes that are currently available, and appends them to another byte array holding all the bytes I've accumulated so far. When it has enough bytes, it "releases" the semaphore, with a parameter of one, which in our case indicates that one packet of reply data is now available. The "ball" is now in the consumer's court.

```
void SendFileWorker::handleReadyRead()
{
    QByteArray bytes = m_serial_port_p->readAll();

    m_rx_bytes_a.append( bytes );
    if ( m_rx_bytes_a.count() == REPLY_PACKET_BUFFER_SIZE )
    {
        m_reply_sem.release( 1 );
    }
}
```

Next, we have the method that waits to "take" the semaphore. When we do this, conceptually this "consumer" code now has taken one "resource," one packet of reply data, and the "producer" code no longer has a resource, until another packet comes in:

```
bool SendFileWorker::receive_rs232_reply( unsigned char * data_p )
{
    bool ret_val = m_reply_sem.tryAcquire( 1, 250 );
    if ( true == ret_val )
    {
        ( void )std::memcpy( data_p, m_rx_bytes_a.constData(),
```

```
            REPLY_PACKET_BUFFER_SIZE );
        m_rx_bytes_a.truncate( 0 );
    }

    return ret_val;
}
```

That seems so simple that it couldn't *not* work, right? That's what I thought! But, in fact, it didn't work at all. The code always waited for the semaphore until it timed out. The data was coming into the serial port, but my code was never receiving it.

This is because **QSerialPort** doesn't *really* support asynchronous sending and receiving using callback functions the way they are usually implemented. Qt uses a somewhat antiquated mechanism called "signals and slots." Signals and slots are *very* useful ways to hook up all kinds of messaging between different pieces of code, and in most cases sending an object a signal is *pretty much* equivalent to calling a method of the object; slots are in fact just methods.

But not *just* like calling a method of the object.

The signals that **QSerialPort** provides to indicate that bytes are ready to receive are really sent when an *event loop* detects a condition and triggers the call to the "slot" method. And that only happens if the event loop runs. And the event loop runs *synchronously*. Doing anything that blocks the thread your Qt object is running in, such as waiting to take a semaphore, will bring the event loop mechanism to a halt, and so that code that is waiting to give the semaphore when it is called will *never be executed*.

It might seem like the logical thing to do is to put the **QSerialPort** object on its own thread, but for various reasons this doesn't really solve the problem. Strangely, **QSemaphore** objects aren't really made to send messages **between** Qt threads. The fundamental method that Qt provides for this is… signals and slots. And your thread won't get those signals if it is is sleeping on a **QSemaphore**, or some kind of queue, or any of the "classic" concurrency programming tools that people who have studied programming formally would expect to use for this purpose.

The sample code available to describe how to use **QSerialPort** does a really, really poor job of explaining this. The asynchronous examples are very contrived and simplified and don't show how to do something that ought to be quite simple: sending a packet of serial data, and waiting on a reply. A quick search of the message boards will reveal that a lot of people have trouble using **QSerialPort** to receive data the way they want it to, and that there is a distinct lack of clear explanations of how to do it.

I got it working, finally, by using something called a local event loop, and it works great. I hook up my signals and slots like so:

```
connect( m_serial_port_p,
```

```
        SIGNAL( errorOccurred( QSerialPort::SerialPortError ) ),
        this,
        SLOT( handleError( QSerialPort::SerialPortError )   )  );
connect( &m_rs232_rx_timer,
        SIGNAL( timeout()                                   ),
        &m_rs232_rx_event_loop,
        SLOT( quit()                                        )  );
connect( m_serial_port_p,
        SIGNAL( readyRead()                                 ),
        &m_rs232_rx_event_loop,
        SLOT( quit()                                        )  );
```

My **QSerialPort** object's **errorOccurred** signal, which includes an error parameter, is now hooked to my object's handleError slot, my **QTimer** object **timeout** signal is hooked up to my local event loop's **quit** slot, and my QSerialPort object's **readyRead** signal is *also* hooked up to my local event loop's **quit** slot. I have to use a timer, because that **readyRead** signal is vague; it just means "at least one byte is available for reading." And I have to keep track of my own timeout condition. I also have to handle retries until I either have the number of bytes I expect, an overflow condition, or I've run out of retries. This is quite a bit uglier and more complicated than just making a single call to wait for a certain number of bytes or a timeout, but it works with perfect reliability now. My receive function now looks something like this:

```
bool SendFileWorker::receive_rs232_reply( unsigned char * data_p )
{
    bool done = false;
    bool overrun = false;
    int tries = 0;
    int max_tries = 5;
    QByteArray packet;

    m_rs232_rx_timer.start( 50 );

    do
    {
        m_rs232_rx_event_loop.exec();

        if ( m_rs232_rx_timer.isActive() )
        {
            /*
                If the event loop exited and the timer is still
                running, we must have gotten the readyRead() signal.
            */
            packet += m_serial_port_p->readAll();

            int len = packet.length();
```

```
            if ( len > REPLY_PACKET_BUFFER_SIZE )
            {
                overrun = true;
            }
            else if ( len == REPLY_PACKET_BUFFER_SIZE )
            {
                ( void )std::memcpy( data_p, packet.constData(),
                    REPLY_PACKET_BUFFER_SIZE );
                done = true;
            }
        }
        else
        {
            tries += 1;
        }

    } while ( ( false == overrun ) &&
              ( false == done ) &&
            ( tries <= max_tries ) );

    return done;

}
```

I can do something similar using the synchronous methods, using **WaitForReadyRead()** instead of using a slot in my own code, but it isn't much simpler. **WaitForReadyRead()** must be allowing my main thread's event loop to run. I tried to take a look at the implementation, to see if it gave me any insight, but it calls another method, and then another, and then some sort of implementor class, using the "pointer to implementor" idiom, also known as "pImpl," and then that implementor calls another method which was defined as a macro, which is a pretty ugly and primitive thing to do in a framework written in C++. Searching for *that* implementation crashed my text editor's "find in files" function as it tried to search through almost 250,000 files of Qt source code.

There's a big difference between "theoretically, you can learn a lot from reading the source code!" to actually being able to read and understand the source code of a project like this, which contains hundreds, if not thousands, of developer-years of work, and at this point, probably a non-zero number of developer-careers, too.

Sometimes you **can** learn a great deal from reading a framework's source code. The PowerPlant framework was written by one very smart guy and it was incredibly readable. I really miss using a framework that was so simple and clear. But Qt is not nearly as easy to understand. And what I am trying to do isn't really very complex or unusual, so the code to do it shouldn't need to be over-complicated.

Anyway.

It's Time to Stroke My Long White Beard and Complain about Kids Today

I first did event-driven programming around 1985, when I was learning how to program the original Macintosh. Event-driven programming has a lot of advantages on small and slow systems. But even on small microprocessors, like the SAM4S2 chip I'm programming, which has only 64K of RAM and a clock speed that is only a bit faster than the original Macintosh, I am accustomed to using tiny operating systems like FreeRTOS that allow me to run multiple tasks and communicate between them very easily, with seamphores and queues that pretty much just work, with no surprises. The old event queue designs were a hack that allowed very slow chips to behave *almost* like they were running fully multi-threaded operating systems and applications; they aren't *really* all that useful when the chips are fast and it is easy to do things in more "computer science-y" ways.

It's actually really, really easy to write event-driven code using primitive multi-threading constructs like semaphores and queues; FreeRTOS does a brilliant job in showing just how this sort of thing can be done in a portable but very efficient way.

I've used a lot of different C++ frameworks over the years: TurboVision, THINK Class Library, the Microsoft Foundation Class Library, PowerPlant, the Object Windows Library, and others I have no doubt forgotten; I've also used a number of frameworks written for languages *other* than C++, like Dylan, NewtonScript, and Java.

Qt has outlasted pretty much all of the frameworks I mentioned. One reason is that it solves a lot of hard code portability problems. But this "clash of civilizations" — of different programming paradigms, really — at the heart of Qt is really making me wonder if there isn't a better framework out there.

Could I find a framework for Windows programming that is more modern, more consistent, and simpler, and not just bigger and more "modern?" (It would be nice if it was cross-platform; that used to be one of Qt's big selling points, but it has become less important to me now that pretty much everyone we are targeting with our products has access to a PC running Windows, and there isn't a good business case to be made for writing a Macintosh version).

I don't think such a thing exists, unfortunately.
