Subject: Re: About RS232
Posted by Mindtraveller on Sun, 17 Mar 2013 07:56:20 GMT
View Forum Message <> Reply to Message

Let me share some thoughts on serial port communication as industrial automation developer. Critics is welcome.
First of all, the main idea behind all this functionality is sending and receiving some actual data to/from PC. The most common scenario is to send request and receive answer (PC = master, device = slave). We also must understand highly asynchronous nature of all these tasks, because at no circiumstances serial port i/o must be done in common thread (especially when it handles GUI). The next idea is crossplatform code, because our great U++ framework is crossplatfrom too.
The last requirement is the ease of tuning: most of time we need to tune some parameters of serial port "on the field", and we must me able to do it without recompiling our code.
These features put together really professional-grade serial library.
Let's recollect them, dividing by levels of abstraction:

1. Port handling and I/O (lowest level).
Crossplatform code handling system-dependent serial port code.
Done with tiny library called crossplatserial with my little modifications. It has Apache license (which is the same as BSD, AFAIK).
This level creates SerialPort objects from external file or code. So if you need to tune i.e. the baudrate, you just change external file and restart your app.
I attach crossplatserial to this message and you may use it, if it is all that you want.

2. Datagramms <-> values (protocol stuff).
Actually the main code should work with data. Not with actual bytes you send through serial port. I achieved it with my BNF library which is responsible for using protocols.
That is how it works:
You have external file defining a protocol. BNF takes plain bytes and returns Vector<Value> and back. For example: ***modbus   (crc_start byte byte word_be word_be crc_end crc_modbus)
        | (crc_start byte byte byte   word_be crc_end crc_modbus);
***icp_din_clear_request   "$" HEX:2 "C" "\0D";
***icp_din_clear_answer   (("!" arg:1 HEX:2) | ("?" arg:0 HEX:2)) "\0D";
Ok. On this level we've left protocols behind and made possible for SerialPort to work with actual Vector<Value> (not with plain bytes). I repeat that protocol is defined in external file, so if you need to tune it, you just change protocol description in file and restart your app. Extremely useful in some cases "in the field".

3. Asynchrony (highest level).
IMO the ideal approach for asynchrony of serial i/o is using queued threads. This means each thread has it's queue and you put there this thread's callback with custom parameters. That is how threads interact, no syncronization objects needed.
This is done with bazaar/MtAlt package I wrote some time ago.
In spite of this frightening description, the actual use of this is simple. Let's look at real-world example: class SensorNB3000: public SerialPortNotify<CallbackQueue> {/*...*/}; //creates callback queue

void SensorNB3000::RequestMeasure()

```
{
 Vector<Value> args;
 args << ((int) addr.GetData()) << 6 << 50 << 1;
 sensorPort->SendReceiveSleep_(PREPARE_TIMEOUT_NB3000);
 sensorPort->SendReceive_
 (
  "modbus",
  args,
  "modbus",
  RECEIVE_TIMEOUT_NB3000,
  NULL,
  Ptr<SiloSensorNB3000>(this),
  &SensorNB3000::OnMeasure
 );
}

void SensorNB3000::OnMeasure(bool result, Vector<Value> args, void *custom)
{
    //...
}
```

This code means: SensorNB3000 class adds some timeout to sensorPort thread queue and then requests send/receive operation with modbus protocol. Send modbus args are {addr,6,50,1} and the handling routing is OnMeasure. The SensorNB3000::OnMeasure() routine is syncroneously called in the SensorNB3000 thread (no sync is needed in your code!).

Here are the most of the concepts I propose. If you find them too comlicated, you may just use attached library. If you consider using all these levels of abstraction in U++, we may discuss it further.

File Attachments

1) crossplatserial.zip, downloaded 675 times