1. **Basic Comm Test/Debugging Setup:** Try debugging technique using miniterm described at <http://tldp.org/HOWTO/Serial-Programming-HOWTO/x56.html#AEN58>
   1. Obtain copy of miniterm: <ftp://sunsite.unc.edu/pub/Linux/docs/LDP/programmers-guide/lpg-0.4.tar.gz>
      1. You may have to compile it—instructions say to check the line #define MODEMDEVICE “/dev/ttyS0” (for COM1, “/dev/ttyS1” for COM2)
   2. If possible to connect two linux computers to test serial comms, connect using a null modem cable – what you enter on one miniterm should show up on the other and vice versa.
   3. If not possible to use two computers, but you can use two ports on a single computer, *“run two miniterms off two virtual consoles. If you free a serial port by disconnecting the mouse, remember to redirect /dev/mouse if it exists. If you use a multiport serial card, be sure to configure it correctly.”*
   4. If neither of these options will work, then we can still try the loopback on a single port. I would think this could still be tested using miniterm.
2. **Out-of-the-Box Linux Serial Comm Functions:** Once you can confirm basic serial comms is working in step 1, try basic custom programs to figure out how to control the comm port in code. *“Don't forget to give the appropriate serial ports the right permissions (e. g.: chmod a+rw /dev/ttyS1)!”*This site has man pages (as unhelpful as they are) for Linux system functions like the file open that also works for opening a serial port for comms: <http://linux.die.net/man/2/open>
   1. **Port Settings Caution –** There are port settings that can cause undesired or unexplained behavior—you need to know what mode you want to operate in (see options below). Start with using the settings in the example code below, and modify if needed for your situation. Details at “[man termios(3](http://man7.org/linux/man-pages/man3/termios.3.html))”  
      Some settings might just be baffling if you didn’t know: *“The devices /dev/ttyS\* are intended to hook up terminals to your Linux box, and are configured for this use after startup. This has to be kept in mind when programming communication with a raw device. E.g. the ports are configured to echo characters sent from the device back to it, which normally has to be changed for data transmission.”*
   2. **Canonical Input Processing –** a read will return only with a new-line (NL or LF), end-of-file (EOF), or end-of-line (EOL) character. I think this can be useful for message-based communications rather than continuous streams. *“A CR (the DOS/Windows default end-of-line) will not terminate a line with the default settings.”* (i.e., CR has to be mapped to LF for a line to be processed). In this mode, your writer has to send the NL for the receiver to receive and process the message. See example code: <http://tldp.org/HOWTO/Serial-Programming-HOWTO/x115.html#AEN125>
   3. **Non-Canonical Input Processing –** in this mode, reads are not completed and processed based on the NL terminator character, but instead use a fixed number of characters or a timeout, or a combination of number of characters and timeouts to control when a read completes. *“This mode should be used if your application will always read a fixed number of characters, or if the connected device sends bursts of characters.”* See example code: <http://tldp.org/HOWTO/Serial-Programming-HOWTO/x115.html#AEN129> (example code section also provides details on the effects of various options created by combinations of *“Two parameters control the behavior of this mode: c\_cc[VTIME] sets the character timer, and c\_cc[VMIN] sets the minimum number of characters to receive before satisfying the read.”*)
   4. **Asynchronous Input –** Both canonical and non-canonical can be used in synchronous and asynchronous modes. *“Synchronous is the default, where a read statement will block, until the read is satisfied. In asynchronous mode the read statement will return immediately and send a signal to the calling program upon completion. This signal can be received by a signal handler.”* Asynchronous doesn’t complete the read when you call it, but rather is set up to spawn a separate thread of execution using a callback (interrupt handler) when the read conditions are completed. Asynch is more complicated to program, but more efficient processing. Synchronous has to wait for the sender to complete sending a message, and causes your code to essential halt until the read condition is met. See example code: <http://tldp.org/HOWTO/Serial-Programming-HOWTO/x115.html#AEN144>
   5. **Waiting for Input from Multiple Sources –** This is a bit more advanced, but might be needed when you put your full architecture together. *“The program example given below will wait for input from two different input sources. If input from one source becomes available, it will be processed, and the program will then wait for new input.   
      The approach presented below seems rather complex, but it is important to keep in mind that Linux is a multi-processing operating system. The select system call will not load the CPU while waiting for input, whereas looping until input becomes available would slow down other processes executing at the same time.”* See example code: <http://tldp.org/HOWTO/Serial-Programming-HOWTO/x115.html#AEN148>
3. **Libserial –** Itmay be possible to debug your libserial code using the miniterm approach above along with some of the clues of modes and settings in the discussion above. This approach seems to be using C code rather than C++, and it sounds like libserial is a C++ wrapper for serial comm. The out-of-the-box, native Linux stuff is simpler, and I would try debugging there first with simple programs. We can then figure out how to move that into the C++ architecture with ROS.