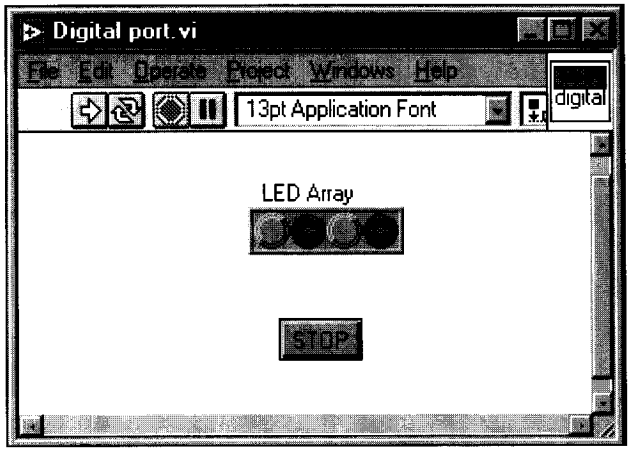
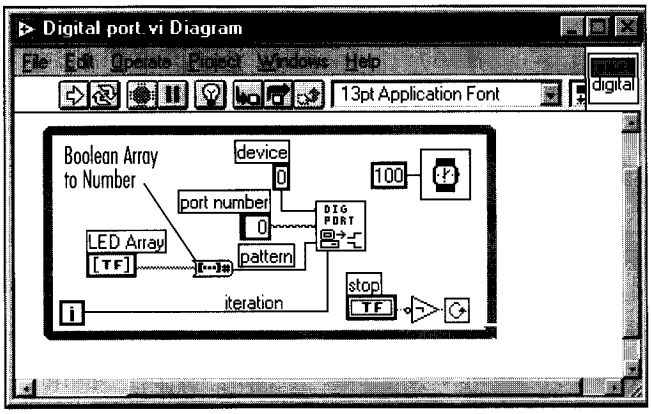


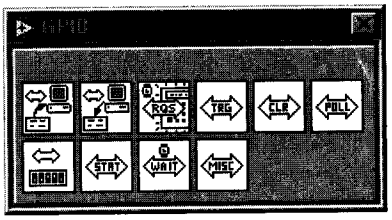
The solution is fairly simple—but is made easier if you find the **Boolean Array to Number** function on the **Boolean** palette. You can take the Boolean array and, using this function to convert the binary value of the Booleans in the array to a decimal number, wire the **pattern** input painlessly.

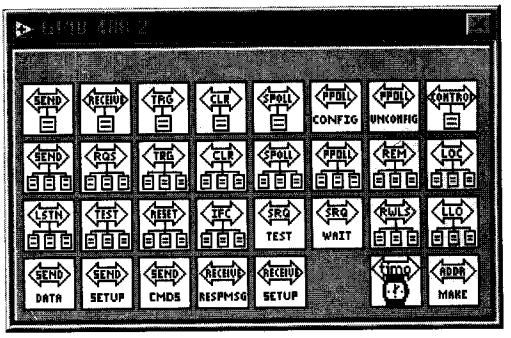


You'll notice there are many other functions in the **Digital I/O** palette. For more advanced applications, consult the LabVIEW manuals.

The GPIB VIs

We could devote a whole chapter or even a book to the subject of GPIB (remember, the GP stands for "general purpose"). Nonetheless, we will skim over the basics to help you get started.



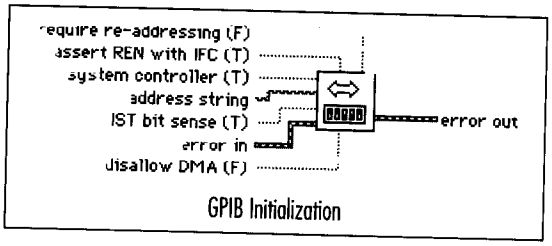


First of all, you'll notice there are *two* palettes for GPIB communication: one is called **GPIB** and the other, chock-full of more functions, **GPIB 488.2**. A word of explanation is due here: GPIB 488.2, also called IEEE 488.2 is the latest "version" of the standard. 488.2 establishes much tighter rules about the communications and command sets of instruments in an attempt to make programming easier and more uniform for different GPIB devices. Most GPIB devices will still work if you use the **GPIB** palette. If you need to adhere to the IEEE 488.2 standard, though, you should use the **GPIB 488.2** palette.

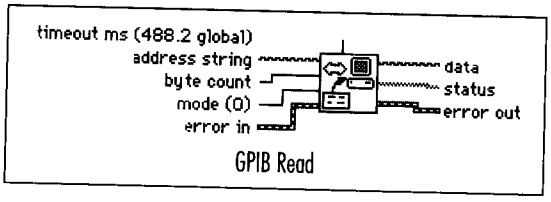
*GPIB VIs will eventually be replaced by the functions in the **VISA** palette. VISA VIs are more general-purpose functions that can communicate with other instrument types, such as VXI. For more information on **VISA** VIs, consult the LabVIEW manuals.*

Remember, all GPIB instruments have an *address*, which is a number between 0 and 30 uniquely identifying the instrument. Sometimes instruments will also have a *secondary address*. In all the GPIB VIs, you must provide the address of the instrument (in string format) in order to talk to it. This makes it easy to use the same VIs to talk to different devices on the same GPIB bus; you can enclose the addresses in a Case Structure, for example.

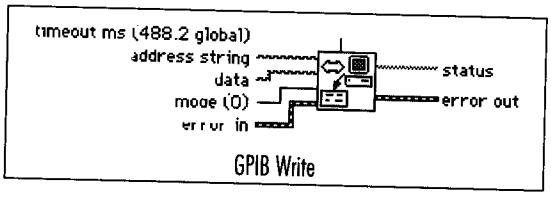
The majority of GPIB communication involves initializing, sending data commands, perhaps reading back a response, triggering the instrument, and closing the communication. With the following subset of simple GPIB VIs from the **GPIB** palette, you can do quite a bit.



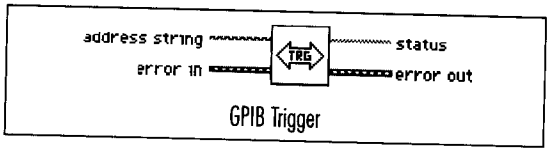
Initializes the GPIB device whose address is given in **address string.** As in DAQ channels, the address is specified as a string because occasionally it may contain nonnumeric characters.



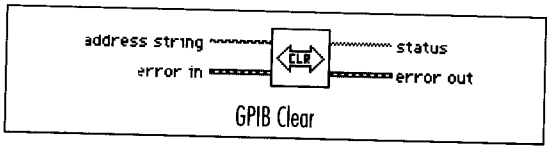
Reads data from the device referenced by **address string**. The number of bytes read is determined by **byte count**. The data read is returned in the **data** string.



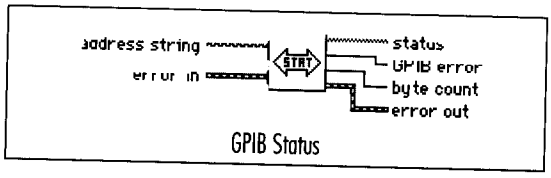
Writes string **data** to the device.



Sends a trigger command to the device.

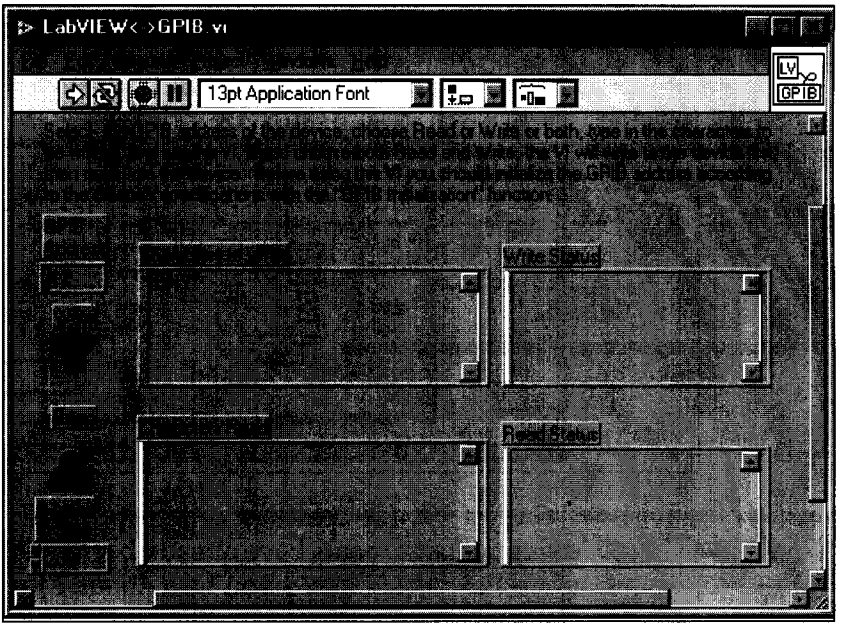


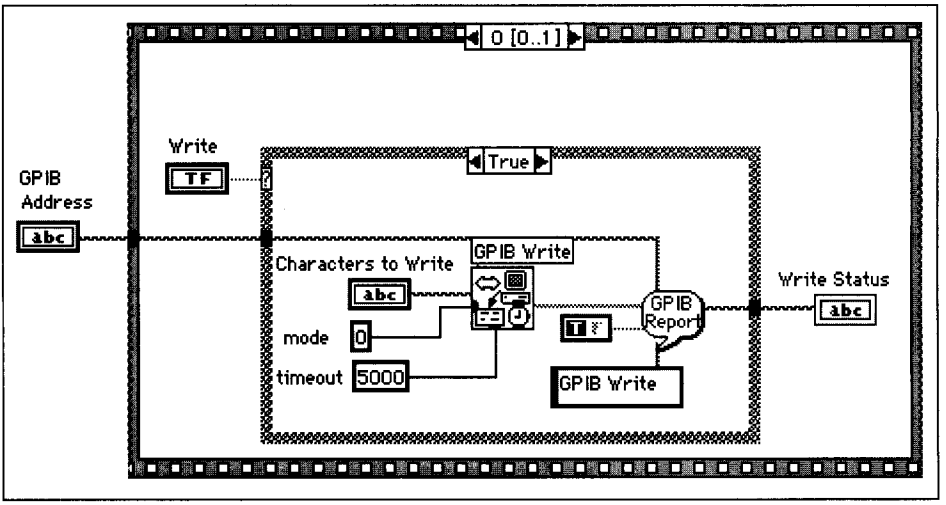
Clears the device.



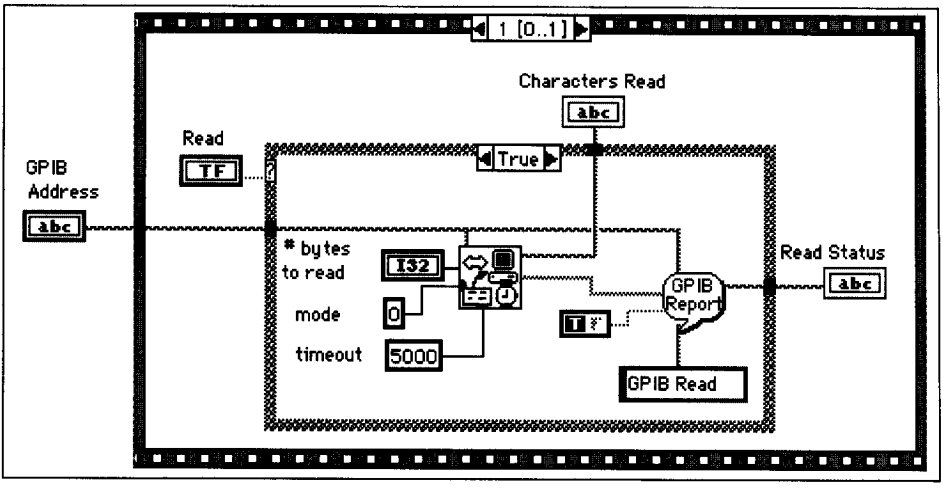
Returns the status of the GPIB controller indicated by **address string** and number of bytes sent in the previous GPIB operation.

A good way to test if you are communicating over GPIB with your instrument is to use the built-in example, **LabVIEW<->GPIB.vi** (found in `examples\instr\smplgpib.llb`) in the full version of LabVIEW.



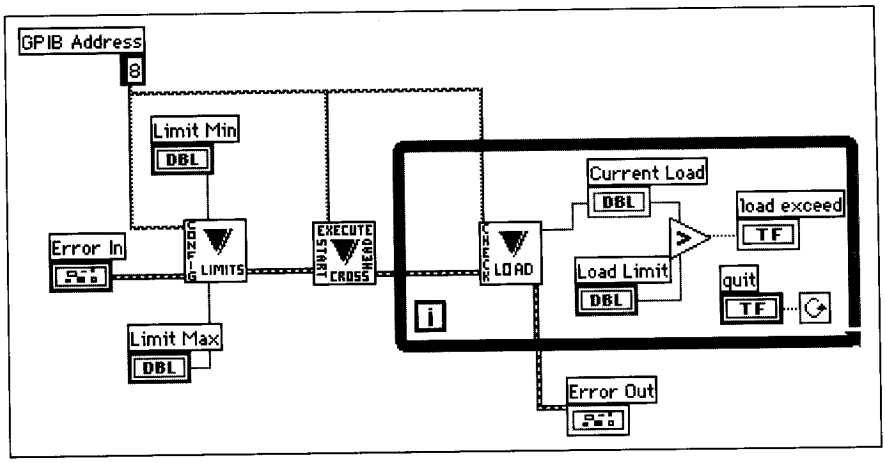


The **GPIB Report** VI in this block diagram is a useful utility VI that reports the status and errors (if any) of the previous GPIB operation. The Boolean input determines whether a dialog box is shown to the user in case of an error. You can find this VI in `examples\instr\smplgpib.llb` in the full version of LabVIEW.

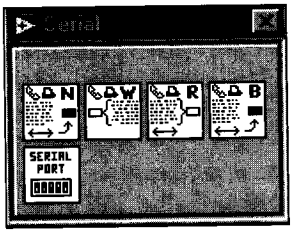


Sets of VIs for communicating specifically with hundreds of different instruments have already been written. These are the famous *instrument drivers* you hear about. Most of them are free of charge; you can get them on the LabVIEW Full Development CD from National Instruments or you can download them over the Internet. Other, more specialized or custom instrument drivers are available from various companies such as Alliance members (3rd party developers endorsed by National Instruments). Sometimes the instrument manufacturer will have the LabVIEW instrument driver available. Instrument drivers can save you hundreds of hours in programming time because the low-level GPIB code has already been taken care of, including specific commands and codes required by the particular instrument.

Often you will only use a few of the VIs included in an instrument driver, since you may only need to manipulate the functions pertinent to your application. One commercial instrument driver, **VI Strength** (available from VI Technology, Austin, Texas), is designed to control Instron Materials Testers through GPIB. The following diagram shows a simple application using just three of the VIs included with **VI Strength**.



Serial VIs



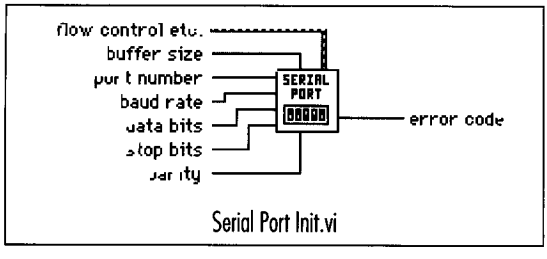
In one sense, serial communication is the simplest to program: after all, only five serial VIs exist on the **Serial** palette. On the other hand, serial communication suffers from abused and ignored hardware standards, obscure and complex programming protocols, and relatively slow data transfer speeds. The difficulties people encounter with writing an application for a serial instrument are rarely LabVIEW-related! We'll take a quick look at how to use the serial port VIs to communicate with a serial device.

You should become familiar with some basic concepts of how serial communication works if you've never used it before. If you have used a modem, and know what things like baud rate, stop bits, and parity roughly mean, then you should know enough to get started. Otherwise, it might be a good idea to read some documentation about the serial port (any good book on RS-232).

From LabVIEW's point of view, serial communication can occur at any specified *port* on the computer. The **port number** has different meanings depending on your platform:

		UNIX	
Windows	MacOS	Solaris 1	Solaris 2
0: COM1	0: modem port	0: /dev/ttya	0: /dev/cua/a
1: COM2	1: printer port	1: /dev/ttyb, etc.	1: /dev/cua/b, etc.
2: COM3			
. . .			
8: COM9			
10: LPT1			
11: LPT2, etc.			

Following is a brief overview of what each serial communication VI does:



Initializes the selected port to the specified settings. Often you can leave inputs unwired if you use the default values which are:

port number: 0

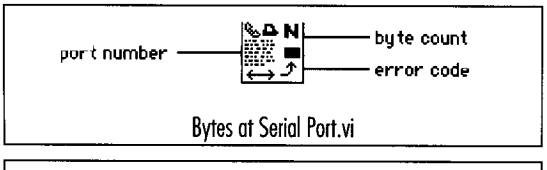
baud rate: 9600

data bits: 8

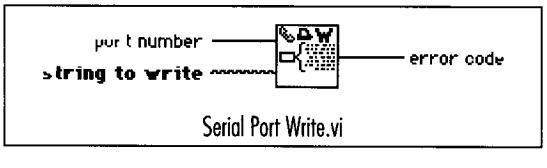
stop bits: 1

parity: none

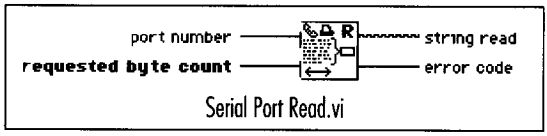
The **flow control** input is a cluster of several different handshaking modes. By default, this VI uses no handshaking. Open the front panel of this VI and copy this complex cluster if you need to use it.



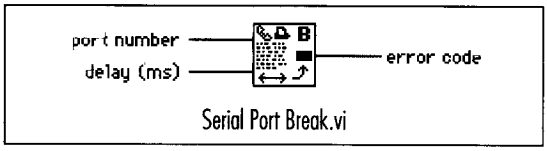
Returns the number of bytes currently at the input buffer of the designated serial port.



Writes the data in the input string to the specified serial port.



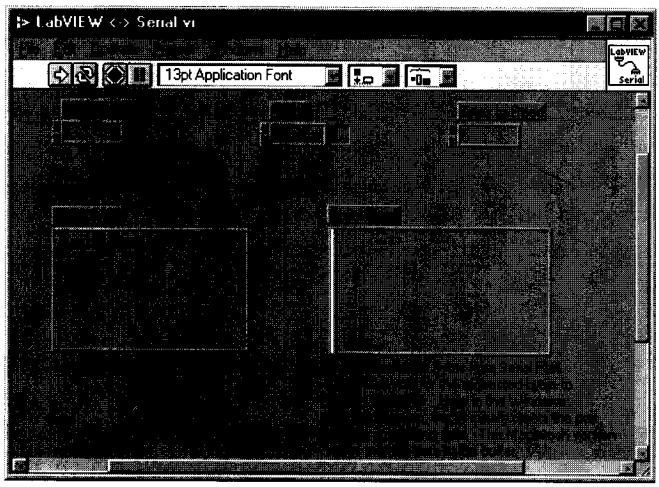
Reads the number of characters specified by **requested byte count** from the specified serial port.



Sends a break to the specified serial port for at least as long as **delay (ms)**.

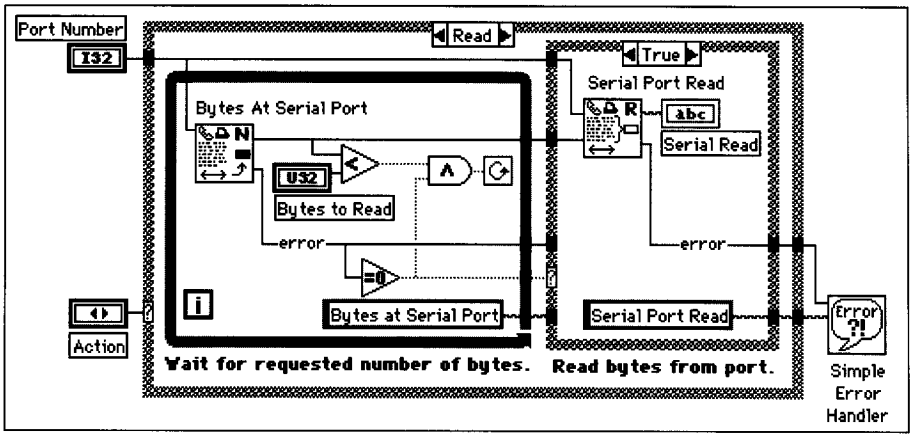
One serial byte corresponds to one ASCII character, which is the way most serial communication is done. Another useful concept to remember is that your computer (and possibly your serial device as well) has a *serial port buffer*. The FIFO (first in, first out) buffer collects the characters as they come in over the serial line. This is important to know, because when you perform a serial read, you are reading the oldest data still in the buffer. Once the data is read, it is discarded from the buffer.

Looking at the above VIs, the sequence for serial communication should appear fairly obvious: initialize, write a command, read the response, repeat. The example included with the full version LabVIEW, **LabVIEW<->Serial Port.vi** (found in `examples\instr\smplserl.llb`) is a great starting place for writing a serial communication VI.

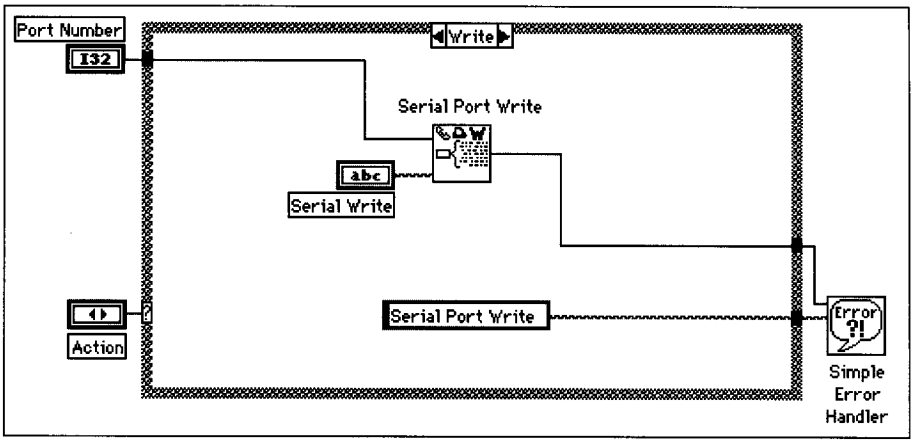


Notice that this VI can read or write to the serial port, but not at the same time. To read, it waits until it finds the expected number of bytes at the serial port buffer. It's a wise move to wait for

the expected number of bytes to make sure all the data from the serial device has arrived—otherwise you'll get erroneous results. However, if you specify a read of a larger number of bytes than actually arrive, your VI will wait indefinitely.



Writing is even easier; just send the string out the serial port:



Some of the following guidelines may help you when writing an application that communicates with a serial instrument:

1. Be absolutely certain that the configuration of the serial port on your instrument matches *exactly* the configuration in your **Serial Port Init** function (baud rate, handshaking, etc.). If even one of these parameters doesn't match, they won't communicate.

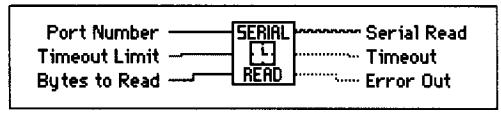
2. Many serial devices expect a carriage return and/or a linefeed character after each string command. LabVIEW doesn't automatically append one, as many terminal emulators do. To add one or

both of these characters to your command string, pop up on the string terminal and choose **Enable '\' Codes**. Then type the carriage return (`\r`) and/or linefeed (`\n`).

3. Use **Bytes At Serial Port!** This VI is very useful. For example, if you didn't know ahead of time how many characters to expect from a serial device, you could use this VI to find out, so you can read the correct amount. You can wire the output of this VI directly to the **requested byte count** on **Serial Port Read.**

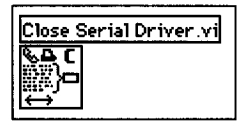
4. If in doubt, flush the buffer before issuing a command to which you expect a response. There is no inherent way to know if the serial data you read with **Serial Port Read** just came in or has been sitting there for hours. To flush the buffer, perform a read of all the bytes in the buffer, and discard the data.

5. There is a very useful example VI called **Serial Read with Timeout.**



This VI can be found in `LabVIEW\examples\instr\smplserl.llb` in the full version of LabVIEW. This function works just like **Serial Port Read,** except that it lets you specify a timeout if the serial instrument doesn't respond. Very useful when your program "hangs" waiting for a response at the serial port.

6. After LabVIEW has called any of the serial VIs, it does not normally "release" the serial port until you quit LabVIEW. This means that no other application (your modem, for example) can access the same serial port unless you first close the serial port (not the same as a serial port break). For some unexplained reason, you don't see a "serial port close" VI on the serial palette. Here's a secret: you can find this **Close Serial Driver** VI,



in the following location: `LabVIEW\vi.lib\_sersup.llb\Close Serial Driver.vi` in the full version of LabVIEW. You have to call up this VI by using the **Select a VI...** We recommend adding it to the **Serial** palette if you find that you need to use it often.