When you design a measurement system, you choose the appropriate hardware and incorporate it into the test program. You have several hardware choices: traditional test instruments, data-acquisition devices, modular instruments, and proprietary systems. In many cases, data-acquisition devices offer measurement capabilities similar to those of traditional instruments but differ in flexibility and features. The programming styles for the devices also differ. Before you select hardware, you should understand these differences and the benefits of each kind of device.

Data acquisition is the process of collecting and measuring electrical signals from sensors, transducers, and test probes or fixtures and bringing them into a computer for processing. Data acquisition also includes the output of analog or digital control signals. It usually involves acquiring the data in hardware and leveraging the PC for data analysis and presentation. As a result, data-acquisition applications are highly customizable, they take advantage of the continuing advances in PC performance and speed, and you can easily modify them to adapt to changing application needs.

Data acquisition devices are programmed through register access to the various chips and components on the hardware and by low-level calls to the operating system to control the bus, map hardware

Plug-in data-acquisition devices are available in PCI, PXI (PCI extensions for instrumentation), and PCMCIA (now better known as PC-card) form factors. External data-acquisition devices connect to PCs via USB, IEEE 1394, and Ethernet. In all of these bus-based examples, the software sends register commands over the bus to the PC to control the device and to transfer data. This tight integration with standard PC technologies allows for similar hardware and software architectures across a range of devices.

Signal conditioning is an important part of most data-acquisition systems. Signal conditioning expands the general-purpose, voltage-measurement nature of data acquisition by adding filtering, amplification, and excitation. These functions, which are usually performed in conditioning modules that connect to data-acquisition devices, allow data-acquisition systems to measure temperature, low or high voltages, strain, pressure, linear displacement, and virtually any other physical variable that sensors can convert to electrical signals.

Programming data-acquisition devices

Data-acquisition devices are programmed through register access to the various chips and components on the hardware and by low-level calls to the operating system to control the bus, map hardware
windows, and allocate PC memory. Nearly all data-acquisition devices include high-level driver software, which isolates users from this low-level programming. The driver software allows users to configure the hardware so that the operating system recognizes it and provides a basic programming interface.

The basic programming flow for a data-acquisition device follows an acquire, analyze, and present model (Figure 1). The device acquires the signal during the acquire step, which breaks down into three substeps. In the configure step, the user selects channels, input ranges, and buffer sizes. In the start step, the user instructs the device to begin acquiring data at a certain rate when it receives a trigger. Last, in the read step, the user directs the program to transfer the data across the bus into the PC’s program memory. Once in memory, the data is available for user manipulation and analysis that can range from simple averaging to complex frequency-domain operations. The device can then present the measurement results in a numeric, graphical, or tabular format. Some development environments include analysis and presentation capabilities; in other environments, these capabilities are add-ons.

For more than 40 years now, test-system designers have built systems that use a computer to interface with and control traditional test instruments. These automated test systems have evolved over the years, but the main component has been constant: The user connects a computer to an instrument through some common I/O-bus interface.

INSTRUMENT CHARACTERISTICS

For many years, GPIB (general-purpose interface bus, also known as IEEE 488), and serial interfaces (mostly RS-232) have been the most common instrument-I/O interfaces. GPIB was designed for instrument control and is a multidrop (as many as 14 instruments), 8-bit, parallel digital-communications interface that transfers data at rates as fast as 8 Mbytes/sec. You most commonly use, RS-232 serial communication, built into every PC, to control modems and printers, but you also use it for scientific and analytical instruments as well as PLCs (programmable-logic controllers). However, unlike GPIB, an RS-232 interface can connect to and control only one device at a time. Transfer rates are typically less than 20 kbps. Other common commercial buses, such as Ethernet and USB, are emerging as candidates for instrument control. Their appeal is in their increased speed, suitability for distributed measurement capabilities, and widespread availability on newer computers.

Regardless of the bus you use to communicate with them, traditional instruments have historically focused on performing well-defined functions. To a greater or lesser extent, each instrument provides a fixed set of measurement capabilities that can require substantial work to adapt to different tasks. For example, some DMMs (digital multimeters) are designed to measure voltage and current. Without additional equipment, they cannot make other types of measurements, such as resistance or temperature. In addition, most DMMs are designed to make one measurement at a time. Therefore, unless the manufacturer has designed the DMM to make multiple simultaneous measurements, you cannot obtain that capability, or, to do so, you must add specialized hardware and software. Another common characteristic of traditional instruments is that they are usually message-based. In other words, they connect to the host computer via an external bus, and you control them remotely by sending them commands or “messages,” such as “Measure: dc volts.”

Unlike data-acquisition devices, which you program by changing the values of hardware registers, instruments are typically programmed via text commands, or “messages.” You can send these messages to the instrument via I/O-interface drivers. For example, a typical GPIB interface has an associated device driver that allows you to send GPIB commands, such as “ibwrt—interface-bus write” or “ibrd—interface-bus read.” One layer of abstraction above this driver are industry-standard I/O libraries, such as those of VISA (Virtual-Instrument Software Architecture). The VISA library hides from you the details of the specific bus, enabling you to write programs that are indifferent to whether you communicate with devices via a GPIB, a serial interface, or a network that uses TCP/IP.

In recent years, however, more and more instrument users have been using instrument drivers to communicate with their instruments (Figure 2). Instrument drivers provide programmers with a collection of high-level functions that internally perform all the low-level programming tasks. For example, an instrument driver for an oscilloscope typically contains functions to initialize the instrument, configure the vertical and horizontal subsystems (for example,
settings, such as each channel’s scale factor—volts/division and the measurement timebase), configure the triggering subsystem (trigger type and settings), and make measurements. Each of these high-level functions contains several low-level commands that, when you combine them, perform the specified function.

As with data-acquisition programming, instrument-driver programming also typically follows a standard progression of function calls (Figure 3). In communicating with an instrument, you first initialize remote communication with the instrument to set it up to receive remote commands. You then configure the instrument settings required for the specified measurement and make the measurement. Finally, you close communication with the instrument so that other applications can use it. Because ranges, functions, and commands are often specific to particular instrument models, you often must modify your program if you substitute a different model instrument. This situation makes it difficult to leverage work from one project to another.

**TECHNOLOGY EVOLVES**

Both of the two main methods that program designers use to communicate with their hardware offer some advantages, and both are subject to disadvantages. Although these programming methods have been available for a long time, they have evolved in recent years to take advantage of new technology. In some cases, the distinction between them has blurred.

The most quantifiable evolution in both data acquisition and test instrumentation is the increase in hardware capabilities. Data-acquisition devices and systems take full advantage of advances in commercial technology. Over the past decade, the communication industry has steadily driven up the speed and resolution of analog-to-digital converters while also driving down their cost enough that data-acquisition devices can readily use the higher-performance components. Also, because a data-acquisition system inherently uses a PC’s buses, memory, and processing power, advances in these areas immediately upgrade data-acquisition devices.

Instrument manufacturers have also begun to adapt to data-acquisition PC technologies that were previously reserved for office-productivity applications. Several instruments run the standard Windows operating systems and provide connectivity through standard PC buses, such as Ethernet and USB.

A perfect demonstration of this hardware evolution comes in the continued...
The equivalent LabView program using the instrument-control approach is somewhat more complex but still basically simple.

Software Evolves

Whereas software advances are less quantifiable, they are certainly observable. Data-acquisition drivers have grown from a simple code library for accessing registers into powerful, high-level programming interfaces. These drivers are further abstracting the hardware-specific programming details, such as buffers, analog-to-digital conversion, and memory access, to dramatically improve ease of use. These interfaces allow you to concentrate on the measurement task instead of hardware-specific commands. On the instrumentation side, drivers have evolved through standardization efforts, such as VXIplug&play and IVI (Interchangeable Virtual Instruments). These standard drivers make programming instruments more straightforward, enable you to quickly create drivers for virtually any instrument, and provide advanced features, such as instrument interchangeability and simulation.

The true change, however, is how well both data-acquisition software and instrument drivers integrate into virtually all development environments used in design, test, and measurement, such as Visual Basic, Visual C++, and LabView. Instead of providing users with a C DLL or a list of instrument commands, modern software offers Visual Basic programmers an ActiveX control with methods, properties, and events. Visual C++ programmers can access native C++ classes, and LabView developers get a set of virtual instruments that follow the native data-flow model. Advances in measurement-specific tools, such as integrated configuration wizards, custom data
types, code generators, and waveform graphs, are also quickly spreading to emerging environments, such as Visual Studio .NET and other .NET-compliant languages.

In addition, some devices can expose both programming models to the user. For example, you can choose either method to program modular instruments that have characteristics of both traditional test instruments and data-acquisition devices. Finally, the flexibility of software-development environments makes it possible to easily integrate both types of devices into one application.

DIFFERENCES REMAIN

Despite the evolution of both hardware and software, inconsistencies remain. Certain fundamental differences still exist in the terminology of data-acquisition devices and instrumentation. Whereas a data-acquisition programmer thinks of a device number, input limits, and a start or stop trigger, the instrument programmer thinks of an instrument-resource descriptor, vertical range, trigger type, and settings. These differences are part of programming models, and interfaces and are difficult to reconcile or change.

Another key difference is the device architecture. A data-acquisition device is fully integrated with a PC bus, whereas an instrument is a stand-alone device that uses cables to communicate with a PC. Whereas a data-acquisition driver writes directly to control registers, an instrument driver sends messages to a device, which then interprets the messages and writes them to its own registers. The impact of this difference is evident in driver software for modular instruments. Although you program these devices in much the same way as you do stand-alone instruments, the driver architecture is optimized for the high data-transfer rates across the PC bus. The functions for controlling data transfer and measurement will remain different for data acquisition devices and instruments, purely because of differences in the hardware architectures.

Now that you have learned about both types of devices and the different methods for programming each, you will probably find it helpful to look at an example of a common measurement task using each type of hardware. The programming example uses the LabView graphical development environment. A LabView program consists of a front panel that serves as the user interface and a block diagram that graphically wires together functions to create the program flow. The example shows three screens—one of the front panel and two of the block diagram, one of which shows the use of each type of hardware and the corresponding programming method. The block diagrams include additional explanation and documentation.

MEASURING MULTIPLE RESISTANCES

Figures 4, 5, and 6 illustrate a LabView program that performs a series of resistance measurements on a bank of resistors. You can build the front panel nearly identically, regardless of the type of device you use but the block diagram differs from one device to the other. Figure 4 shows the design of the front panel of the program. Except for controls specifying which hardware device to use, and some optional device-specific configuration items, the user interface is identical regardless of the hardware used.

The first block diagram shows how to perform this task using data-acquisition hardware (Figure 5). This example assumes that you are using a multifunction data-acquisition board and uses a relatively simple sequence of functions to configure the board and acquire the signal. Note the use of software averaging of a large number of scans for more accurate results.

In Figure 6, which uses a DMM and a switch to perform the same task, you see some differences emerge. First, making the measurements requires several additional configuration steps. In addition, most data-acquisition devices can acquire measurements on multiple channels, whereas most DMMs are single-channel units that require the use of a switch matrix to route the signal to and from the correct resistors. Notice also that this example does not require averaging, because DMMs typically take higher resolution measurements, although at slower rates.

WHAT’S YOUR BEST APPROACH?

How do you choose the appropriate device for a specific application? The choice comes down to four factors:

- Capabilities: Although the importance of this factor diminishes as device capabilities become more similar, it is obviously important to choose hardware that can make the measurements that the application requires. For example, don’t try to measure 10 fA with a device whose input leakage current is 100 pA.
- Flexibility: How much hardware flexibility does the application require? Are the capabilities of an instrument sufficient or is a less defined, more flexible device desirable?
- Synchronization: Does your application require tight timing and synchronization between a number of measurements? Do you need to easily pass signals between devices across a common backplane?
- Support: What type of support is available for each device under consideration? Does the device offer integrated software for the specific development environment? Does the hardware vendor offer assistance in using the product?

Although the hardware characteristics and capabilities of data-acquisition devices and instruments continue to converge, their inherent programming models remain somewhat divergent. As the differences continue to decrease, the choice depends less on hardware and software restrictions and more on those factors that matter to the designer. Practicality rather than ideology becomes the key to the choice of hardware.

Authors’ biographies

Dany Cheij holds a BSME from Massachusetts Institute of Technology (Cambridge, MA) and an MS in Engineering and Public Policy from Carnegie-Mellon University (Pittsburgh). He is a product manager responsible for marketing of instrument-control products at National Instruments (Austin, TX), where he has worked for four and a half years. He is a member of IEEE and enjoys basketball, running, and music.

Malcolm Borgendale, who holds a BSME from the University of Minnesota (Minneapolis/St Paul), is a product manager responsible for measurement-driver software at National Instruments (Austin, TX). He has worked at the company for three years and lists his leisure activities as cycling, running, and hockey.