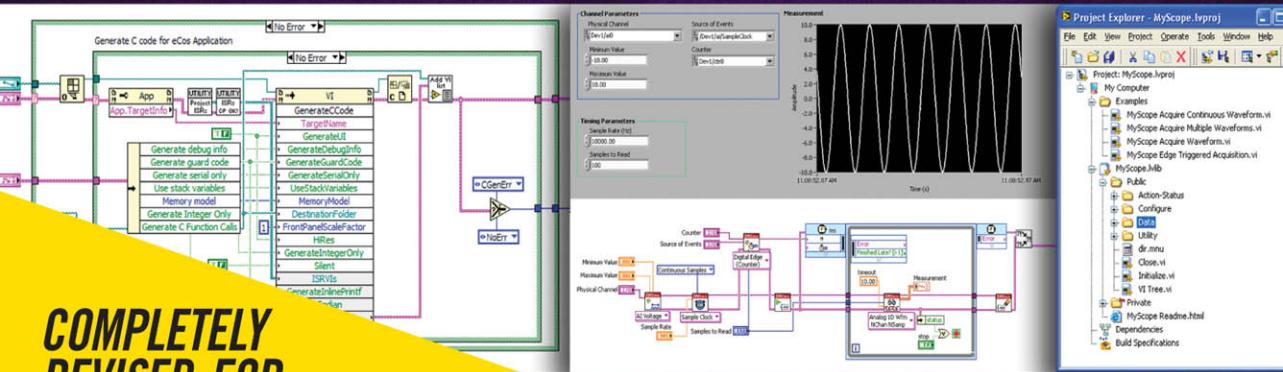


# LabVIEW

# Graphical Programming

FOURTH  
EDITION



**COMPLETELY  
REVISED FOR  
LabVIEW 8.0**

*Covers the  
CLAD & CLD  
Exams*

Gary W. Johnson  
Richard Jennings

---

# **LabVIEW Graphical Programming**

## **Fourth Edition**

**Gary W. Johnson**

**Richard Jennings**

**McGraw-Hill**

New York Chicago San Francisco Lisbon London  
Madrid Mexico City Milan New Delhi San Juan  
Seoul Singapore Sydney Toronto

Copyright © 2006 by The McGraw-Hill Companies. All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

ISBN: 978-0-07-150153-8

MHID: 0-07-150153-3

The material in this eBook also appears in the print version of this title: ISBN: 978-0-07-145146-8, MHID: 0-07-145146-3.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

McGraw-Hill eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative please visit the Contact Us page at [www.mhprofessional.com](http://www.mhprofessional.com).

Information has been obtained by McGraw-Hill from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill, or others, McGraw-Hill does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

#### TERMS OF USE

This is a copyrighted work and The McGraw-Hill Companies, Inc. ("McGraw-Hill") and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." McGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

---

# Contents

Preface .....	xi
Acknowledgments .....	xv
<b>Chapter 1 Roots .....</b>	<b>1</b>
LabVIEW and Automation .....	2
Virtual instruments: LabVIEW's foundation .....	4
Why use LabVIEW? .....	7
The Origin of LabVIEW .....	8
Introduction .....	9
A vision emerges .....	9
All the world's an instrument .....	11
A hard-core UNIX guy won over by the Macintosh .....	12
Putting it all together with pictures .....	12
Favoring the underdog platform for system design .....	15
Ramping up development .....	15
Stretching the limits of tools and machine .....	16
Facing reality on estimated development times .....	18
Shipping the first version .....	19
Apple catches up with the potential offered by LabVIEW .....	19
LabVIEW 2: A first-rate instrument control product becomes a world-class programming system .....	22
The port to Windows and Sun .....	23
LabVIEW 3 .....	24
LabVIEW 4 .....	25
LabVIEW branches to BridgeVIEW .....	26
LabVIEW 5 .....	26
The LabVIEW RT branch .....	28
LabVIEW 6 .....	29
LabVIEW 7 .....	29
LabVIEW 8 .....	31
Crystal Ball Department .....	32
LabVIEW influences other software products .....	32
LabVIEW Handles Big Jobs .....	34
<b>Chapter 2 Getting Started .....</b>	<b>37</b>
About the Diagrams in This Book .....	37
Sequencing and Data Flow .....	38

LabVIEW under the Hood .....	39
The parts of a VI .....	40
How VIs are compiled .....	41
Multitasking, multithreaded LabVIEW .....	41
The LabVIEW Environment .....	43
Front panel .....	44
Controls .....	45
Property nodes .....	45
Block diagram .....	48
SubVIs .....	48
Icons .....	49
Polymorphic VIs .....	50
Data .....	50
Clusters .....	50
Typedefs .....	52
Arrays .....	53
Debugging .....	54
See what the subVIs are up to .....	54
Peeking at data .....	55
One step at a time .....	55
Execution highlighting .....	57
Setting breakpoints .....	57
Suspend when called .....	58
Calling Other Code .....	58
CINs .....	58
Dynamic link libraries .....	59
Programming by Plagiarizing .....	59
Bibliography .....	60
<b>Chapter 3 Controlling Program Flow .....</b>	<b>61</b>
Sequences .....	61
Data Dependency .....	62
Adding Common Threads .....	63
Looping .....	64
While LOOPS .....	65
For Loops .....	66
Shift registers .....	67
Uninitialized shift registers .....	69
Globals .....	71
Global and local variables .....	71
Built-in global variables—and their hazards .....	73
Local variables .....	75
Events .....	81
Notify and Filter events .....	81
Mechanical actions .....	85
Dynamic events .....	85
Design Patterns .....	87
Initialize and then loop .....	87
Independent parallel loops .....	89
Client-server .....	90
Client-server (with autonomous VIs) .....	92
State machines .....	94
Queued message handler .....	98
Event-driven applications .....	100
Bibliography .....	102

<b>Chapter 4 LabVIEW Data Types .....</b>	<b>103</b>
Numeric Types .....	104
Strings .....	105
Building strings .....	106
Parsing strings .....	107
Dealing with unprintables .....	110
Spreadsheets, strings, and arrays .....	110
Arrays .....	114
Initializing arrays .....	117
Array memory usage and performance .....	119
Clusters .....	122
Waveforms .....	125
Data Type Conversions .....	127
Conversion and coercion .....	128
Intricate conversions and type casting .....	129
Flatten To String (... Do what?) .....	132
Enumerated types (enums) .....	133
Get Carried Away Department .....	134
<b>Chapter 5 Timing .....</b>	<b>137</b>
Where Do Little Timers Come From? .....	137
Using the Built-in Timing Functions .....	138
Intervals .....	139
Timed structures .....	140
Timing sources .....	141
Execution and priority .....	143
Timing guidelines .....	145
Sending timing data to other applications .....	146
High-resolution and high-accuracy timing .....	147
Bibliography .....	149
<b>Chapter 6 Synchronization .....</b>	<b>151</b>
Polling .....	152
Events .....	153
Occurrences .....	155
Notifiers .....	158
Queues .....	160
Semaphores .....	161
Me and You, Rendezvous .....	165
<b>Chapter 7 Files .....</b>	<b>167</b>
Accessing Files .....	167
File Types .....	169
Writing Text Files .....	170
Reading Text Files .....	172
Formatting to Text Files .....	175
Binary Files .....	176
Writing binary files .....	178
Reading binary files .....	179
Writing Datalog Files .....	181
Reading Datalog Files .....	183
Datalog file utilities .....	183

<b>Chapter 8 Building an Application .....</b>	<b>185</b>
Define the Problem .....	186
Analyze the user's needs .....	186
Gather specifications .....	187
Draw a block diagram .....	189
Specify the I/O Hardware .....	191
Prototype the User Interface .....	192
Panel possibilities .....	193
First Design and Then Write Your Program .....	196
Ask a Wizard .....	197
Top-down or bottom-up? .....	197
Modularity .....	198
Choose an architecture: Design patterns .....	200
The VI hierarchy as a design tool .....	201
Sketching program structure .....	202
Pseudocoding .....	203
Ranges, coercion, and default values .....	204
Handling errors .....	207
Putting it all together .....	213
Testing and Debugging Your Program .....	214
Tracing execution .....	214
Checking performance .....	216
Final Touches .....	217
VBL epilogue .....	218
Studying for the LabVIEW Certification Exams .....	218
CLAD .....	218
CLD .....	221
Example 1: Traffic light controller .....	223
Example 2: Car wash controller .....	224
Example 3: Security system .....	227
Bibliography .....	229
<b>Chapter 9 Documentation .....</b>	<b>231</b>
VI Descriptions .....	231
Control Descriptions .....	232
Custom Online Help .....	233
Documenting the Diagram .....	234
VI History .....	234
Other Ways to Document .....	235
Printing LabVIEW Panels and Diagrams .....	235
Putting LabVIEW screen images into other documents .....	236
Writing Formal Documents .....	238
Document outline .....	238
Connector pane picture .....	239
VI description .....	239
Terminal descriptions .....	240
Programming examples .....	241
Distributing Documents .....	241
<b>Chapter 10 Instrument Driver Basics .....</b>	<b>243</b>
Finding Instrument Drivers .....	243
Driver Basics .....	245
Communication standards .....	245
Learn about Your Instrument .....	249
Determine Which Functions to Program .....	250

Establish Communications .....	251
Hardware and wiring .....	252
Protocols and basic message passing .....	254
Bibliography .....	256
<b>Chapter 11 Instrument Driver Development Techniques .....</b>	<b>257</b>
Plug-and-Play Instrument Drivers .....	259
General Driver Architectural Concepts .....	260
Error I/O flow control .....	261
Modularity by grouping of functions .....	265
Project organization .....	266
Initialization .....	267
Configuration .....	268
Action and status .....	270
Data .....	270
Utility .....	271
Close .....	272
Documentation .....	273
Bibliography .....	274
<b>Chapter 12 Inputs and Outputs .....</b>	<b>275</b>
Origins of Signals .....	275
Transducers and sensors .....	276
Actuators .....	278
Categories of signals .....	279
Connections .....	284
Grounding and shielding .....	284
Why use amplifiers or other signal conditioning? .....	291
Choosing the right I/O subsystem .....	299
Network everything! .....	303
Bibliography .....	304
<b>Chapter 13 Sampling Signals .....</b>	<b>305</b>
Sampling Theorem .....	305
Filtering and Averaging .....	307
About ADCs, DACs, and Multiplexers .....	309
Digital-to-analog converters .....	314
Digital codes .....	315
Triggering and Timing .....	316
A Little Noise Can Be a Good Thing .....	317
Throughput .....	319
Bibliography .....	321
<b>Chapter 14 Writing a Data Acquisition Program .....</b>	<b>323</b>
Data Analysis and Storage .....	325
Postrun analysis .....	326
Real-time analysis and display .....	337
Sampling and Throughput .....	343
Signal bandwidth .....	343
Oversampling and digital filtering .....	344
Timing techniques .....	350
Configuration Management .....	350
What to configure .....	351
Configuration editors .....	352

Configuration compilers .....	362
Saving and recalling configurations .....	365
A Low-Speed Data Acquisition Example .....	370
Medium-Speed Acquisition and Processing .....	373
Bibliography .....	375
<b>Chapter 15 LabVIEW RT .....</b>	<b>377</b>
Real Time Does Not Mean Real Fast .....	377
RT Hardware .....	379
Designing Software to Meet Real-Time Requirements .....	382
Measuring performance .....	383
Shared resources .....	388
Multithreading and multitasking .....	389
Organizing VIs for best real-time performance .....	391
Context switching adds overhead .....	393
Scheduling .....	395
Timed structures .....	395
Communications .....	398
Bibliography .....	399
<b>Chapter 16 LabVIEW FPGA .....</b>	<b>401</b>
What Is an FPGA? .....	401
LabVIEW for FPGAs .....	403
RIO hardware platforms .....	403
Plug-in cards .....	403
CompactRIO .....	404
Timing and synchronization .....	405
Compact Vision .....	405
Application Development .....	406
Compiling .....	406
Debugging .....	408
Synchronous execution and the enable chain .....	408
Clocked execution and the single-cycle Timed Loop .....	411
Parallelism .....	413
Pipelining .....	413
Conclusions .....	414
Bibliography .....	415
<b>Chapter 17 LabVIEW Embedded .....</b>	<b>417</b>
Introduction .....	417
History .....	417
LabVIEW Embedded Development Module .....	419
The technology: What's happening .....	419
Running LabVIEW Embedded on a new target .....	421
Porting the LabVIEW runtime library .....	422
Incorporating the C toolchain .....	423
The Embedded Project Manager .....	424
LEP plug-in VIs .....	425
Target_OnSelect .....	426
Other plug-in VIs .....	426
Incorporating I/O drivers .....	429
LabVIEW Embedded programming best practices .....	431
Interrupt driven programming .....	434
LabVIEW Embedded targets .....	435

<b>Chapter 18 Process Control Applications .....</b>	<b>437</b>
Process Control Basics .....	438
Industrial standards .....	438
Control = manipulating outputs .....	444
Process signals .....	447
Control system architectures .....	449
Working with Smart Controllers .....	455
Single-loop controllers (SLCs) .....	461
Other smart I/O subsystems .....	463
Man-Machine Interfaces .....	463
Display hierarchy .....	469
Other interesting display techniques .....	473
Handling all those front panel items .....	474
Data Distribution .....	475
Input scanners as servers .....	476
Handling output data .....	477
Display VIs as clients .....	479
Using network connections .....	482
Real-time process control databases .....	484
Simulation for validation .....	485
Sequential Control .....	486
Interlocking with logic and tables .....	486
State machines .....	487
Initialization problems .....	489
GrafcetVIEW—a graphical process control package .....	490
Continuous Control .....	492
Designing a control strategy .....	493
Trending .....	499
Real-time trends .....	499
Historical trends .....	502
Statistical process control (SPC) .....	505
Alarms .....	506
Using an alarm handler .....	508
Techniques for operator notification .....	511
Bibliography .....	512
<b>Chapter 19 Physics Applications .....</b>	<b>513</b>
Special Hardware .....	514
Signal conditioning .....	514
CAMAC .....	518
Other I/O hardware .....	518
Field and Plasma Diagnostics .....	520
Step-and-measure experiments .....	520
Plasma potential experiments .....	527
Handling Fast Pulses .....	533
Transient digitizers .....	533
Digital storage oscilloscopes (DSOs) .....	536
Timing and triggering .....	537
Capturing many pulses .....	539
Recovering signals from synchronous experiments .....	543
Handling Huge Data Sets .....	546
Reducing the amount of data .....	546
Optimizing VIs for memory usage .....	547
Bibliography .....	553

<b>Chapter 20 Data Visualization, Imaging, and Sound .....</b>	<b>555</b>
Graphing .....	556
Displaying waveform and cartesian data .....	558
Bivariate data .....	563
Multivariate data .....	565
3D Graphs .....	570
Intensity Chart .....	571
Image Acquisition and Processing .....	572
System requirements for imaging .....	574
Using IMAQ Vision .....	577
IMAQ components .....	577
Sound I/O .....	586
DAQ for sound I/O .....	586
Sound I/O functions .....	587
Sound input .....	587
Sound output .....	588
Sound files .....	588
Bibliography .....	589
<b>Index .....</b>	<b>591</b>

---

# Preface

Twenty years have passed since the release of LabVIEW. During this period, it has become the dominant programming language in the world of instrumentation, data acquisition, and control. A product of National Instruments Corporation (Austin, Texas), it is built upon a purely graphical, general-purpose programming language, *G*, with extensive libraries of functions, an integral compiler and debugger, and an application builder for stand-alone applications. The LabVIEW development environment runs on Apple Macintosh computers and IBM PC compatibles with Linux or Microsoft Windows. Programs are portable among the various development platforms. The concept of *virtual instruments* (VIs), pioneered by LabVIEW, permits you to transform a real instrument (such as a voltmeter) into another, software-based instrument (such as a chart recorder), thus increasing the versatility of available hardware. Control panels mimic real panels, right down to the switches and lights. All programming is done via a block diagram, consisting of icons and wires, that is directly compiled to executable code; there is no underlying procedural language or menu-driven system.

Working with research instrumentation, we find LabVIEW indispensable—a flexible, time-saving package without all the frustrating aspects of ordinary programming languages. The one thing LabVIEW had been missing all these years was a useful application-oriented book. The manuals are fine, once you know what you want to accomplish, and the classes offered by National Instruments are highly recommended if you are just starting out. But how do you get past that first blank window? What are the methods for designing an efficient LabVIEW application? What about interface hardware and real-world signal-conditioning problems? In this book, we describe practical problem-solving techniques that aren't in the manual or in the introductory classes—methods you learn only by experience. The principles and techniques discussed in these pages are fundamental to the work of a LabVIEW programmer. This is by no means a rewrite of the manuals or other introductory books, nor is it a substitute for a course in

LabVIEW basics. You are encouraged to consult those sources, as well, in the process of becoming a skilled LabVIEW developer.

This fourth edition is founded on LabVIEW 8, but we've worked closely with National Instruments to ensure its relevance now and through future versions of LabVIEW. Chapter 1, "Roots," starts off with an entertaining history of the development of LabVIEW. New to this edition is coverage of material on National Instruments' certification exams. There are three levels of LabVIEW certification: Certified LabVIEW Associate Developer (CLAD), Certified LabVIEW Developer (CLD), and Certified LabVIEW Architect (CLA). Each exam builds on the knowledge required for the previous exam. We have worked closely with National Instruments to highlight study material in this book for the first two certification exams. Throughout Chapters 2 through 9 you will find **CLAD** or **CLD** icons next to sections covered on the certification exams.

In Chapters 2 through 9, we get down to the principles of programming in G. After a discussion of the principles of dataflow programming, we discuss programming structures, data types, timing, synchronization, and file I/O. Chapter 8, "Building an Application," shows you how to design a LabVIEW application. Here we assume that you are not a formally trained software engineer, but rather a technically skilled person with a job to do (that certainly describes us!). We'll walk through the development of a real application that Gary wrote, starting with selection of hardware, then prototyping, designing, and testing the program. Chapter 9, "Documentation," covers this important but oft-neglected topic. We discuss recommended practice for creating effective documentation as it pertains to the world of LabVIEW. If you know the material in Chapters 2 through 9 you should have no problems with the certification exams. At the end of Chapter 8 we provide three practice exams for the Certified LabVIEW Developer (CLD) Exam. Use the knowledge you gained in Chapters 2 through 9 to complete these practice exams in four hours and you are well on your way to certification.

If you connect your computer to any external instruments, you will want to read Chapters 10 and 11, "Instrument Driver Basics" and "Instrument Driver Development Techniques." We begin with the basics of communications and I/O hardware (GPIB, serial, and VXI), then cover recommended driver development techniques and programming practices, especially the virtual instrument standard architecture (VISA) methods. Instrument drivers can be fairly challenging to write. Since it's one of our specialties, we hope to pass along a few tricks.

The basics of interface hardware, signal conditioning, and analog/digital conversion are discussed in Chapter 12, "Inputs and Outputs," and Chapter 13, "Sampling Signals." Notably, these chapters contain no LabVIEW programming examples whatsoever. The reason is

simple: more than half the “LabVIEW” questions that coworkers ask us turn out to be hardware- and signal-related. Information in this chapter is vital and will be useful no matter what software you may use for measurement and control. Chapter 14, “Writing a Data Acquisition Program,” contains a practical view of data acquisition (DAQ) applications. Some topics may seem at first to be presented backward—but for good reasons. The first topic is data analysis. Why not talk about sampling rates and throughput first? Because the only reason for doing data acquisition is to collect data for analysis. If you are out of touch with the data analysis needs, you will probably write the wrong data acquisition program. Other topics in this chapter are sampling speed, throughput optimization, and configuration management. We finish with some of the real applications that you can use right out of the box.

LabVIEW RT brings the ease of graphical programming to the arcane world of real-time system programming. In Chapter 15, “LabVIEW RT,” we show you how LabVIEW RT works and how to achieve top performance by paying attention to code optimization, scheduling, and communications.

When software-timed real-time applications won’t fit the bill, LabVIEW FPGA is the way to go. LabVIEW FPGA applications are not constrained by processor or operating system overhead. With LabVIEW FPGA you can write massively parallel hardware-timed digital control applications with closed loop rates in the tens of megahertz. Chapter 16, “LabVIEW FPGA,” gives a solid introduction to programming FPGAs with LabVIEW.

Embedded computer systems are all around us—in our cars, VCRs, appliances, test equipment, and a thousand other applications. But until now, LabVIEW has not been a viable development system for those miniaturized computers. Chapter 17, “LabVIEW Embedded,” introduces a new version of LabVIEW capable of targeting any 32-bit microprocessor.

Chapter 18, “Process Control Applications,” covers industrial control and all types of measurement and control situations. We’ll look at human-machine interfaces, sequential and continuous control, trending, alarm handling, and interfacing to industrial controllers, particularly programmable logic controllers (PLCs). We frequently mention a very useful add-on toolkit that you install on top of LabVIEW, called the Datalogging and Supervisory Control Module (formerly available as BridgeVIEW), which adds many important features for industrial automation.

LabVIEW has a large following in physics research, so we wrote Chapter 19, “Physics Applications.” Particular situations and solutions in this chapter are electromagnetic field and plasma diagnostics, measuring fast pulses with transient recorders, and handling very large data sets. This last topic, in particular, is of interest to almost all users

because it discusses techniques for optimizing memory usage. (There are tidbits like this all through the book—by all means, read it cover to cover!)

Chapter 20, “Data Visualization, Imaging, and Sound,” shows off some of the data presentation capabilities of LabVIEW. Some third-party products and toolkits (such as IMAQ for imaging) are featured. They enable you to acquire video signals, process and display images, make three-dimensional plots, and record and play sound.

As far as possible, this book is platform-independent, as is LabVIEW itself. Occasional topics arise where functionality is available on only one or two of the computer platforms. The LabVIEW user manual contains a portability guide that you can consult when developing applications that you intend to propagate among various platforms.

Many important resources are available only via the Internet. For your convenience, Internet addresses are interspersed in the text. While writing this book, we found that user-supplied example VIs were hard to obtain, owing to the fact that so many of us work for government laboratories and places that just don’t like to give away their software. Where it was not possible to obtain the actual code, we attempted to reconstruct the important aspects of real applications to give you an idea of how you might solve similar problems.

Third-party LabVIEW products, such as driver and analysis packages, are described where appropriate. They satisfy important niche requirements in the user community at reasonable cost, thus expanding the wide applicability of LabVIEW.

If nothing else, we hope that our enthusiasm for LabVIEW rubs off on you.

*Gary W. Johnson and Richard Jennings*

---

# Acknowledgments

We would like to thank the engineers, developers, and managers at National Instruments who supplied vital information without which this book would not be possible, particularly Jeff Kodosky, David Gardner, Joel Sumner, Newton Petersen, P. J. Tanzillo, and Kathy Brown. A special thanks goes to Zaki Chasmawala of National Instruments for proofreading and highlighting the pertinent parts of Chapters 2 through 9 covered by the certification exams.

Credit also goes to our wives, Katharine Decker Johnson and Patty Jennings, whose patience during this project cannot be overstated. And to Elizabeth, Danny, Chris, and David Jennings—thank you for understanding.

Finally, thanks to our editor, Wendy Rinaldi.

*Gary W. Johnson:  
To my wife, Katharine*

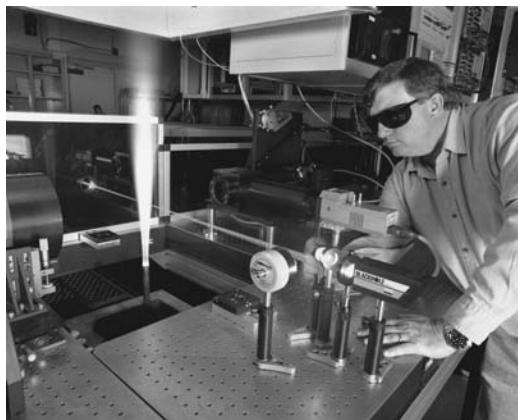
*Richard Jennings:  
To my Lord and Savior, Jesus Christ*

## ABOUT THE AUTHORS

**Gary W. Johnson** is an instrumentation engineer at the Lawrence Livermore National Laboratory. He has a BS degree in electrical engineering/bioengineering from the University of Illinois. His professional interests include measurement and control systems, electro-optics, communications, transducers, circuit design, and technical writing. In his spare time, he enjoys woodworking, bicycling, and amateur radio. He and his wife, Katharine, a scientific illustrator, live in Livermore, California, with their twin Afghan hounds, Chloe and Derby.



*LabVIEW goes aloft.* Gary works on a LabVIEW-based laser wavelength controller for an airborne LIDAR system aboard an Air Force C-135.



*Do not look into laser with remaining good eye.* Richard wisely wears his laser eye protection while manually aligning laser beams through a piloted jet burner in the Turbulent Combustion Laboratory.

**Richard Jennings** is president of Jennings Embedded Services, LLC in San Antonio, Texas. His company was founded in 2005 to serve as a hardware and software resource for companies engaged in embedded software and hardware development in emerging embedded markets such as industrial control, wireless, and embedded instrumentation. He is a 15-year veteran hardware and software engineer. Prior to starting Jennings Embedded Services, a National Instruments Certified Alliance partner, Jennings worked as a system integrator at Sandia National Laboratories and at Lawrence Livermore National Laboratories in Livermore, California. He holds an Associate Degree in Laser-ElectroOptics from Texas State Technical Institute in Waco, Texas, and is a Certified LabVIEW Developer. In 2003 he was awarded National Instruments' prestigious Virtual Instrumentation Professional (VIP) award. [www.jembedded.com](http://www.jembedded.com)

# Roots

LabVIEW has certainly made life easier for Gary, the engineer. I remember how much work it was in the early 1980s, writing hideously long programs to do what appeared to be simple measurement and control tasks. Scientists and engineers automated their operations only when it was absolutely necessary, and the casual users and operators wouldn't dare to tamper with the software because of its complexity. This computer-based instrumentation business was definitely more work than fun. But everything changed when, in mid-1987, I went to a National Instruments (NI) product demonstration. The company was showing off a new program that ran on a Macintosh. It was supposed to do something with instrumentation, and that sounded interesting. When I saw what those programmers had done—and what LabVIEW could do—I was stunned! Wiring up *icons* to write a program? *Graphical* controls? Amazing! I had to get hold of this thing and try it out for myself.

By the end of the year, I had taken the LabVIEW class and started on my first project, a simple data acquisition system. It was like watching the sun rise. There were so many possibilities now with this easy and fun-to-use programming language. I actually started looking for things to do with it around the lab (and believe me, I found them). It was such a complete turnaround from the old days. Within a year, LabVIEW became an indispensable tool for my work in instrumentation and control. Now my laboratories are not just *computerized*, but also *automated*. A computerized experiment or process relies heavily on the human operators—the computer makes things easier by taking some measurements and doing simple things like that, but it's far from being a hands-off process. In an automated experiment, on the other hand, you set up the equipment, press the Start button on the LabVIEW screen, and watch while the computer orchestrates a sequence

of events, takes measurements, and presents the results. That's how you want your system to work, and that's where LabVIEW can save the day. Let's start by taking a look at the world of automation.

## LabVIEW and Automation

Computers are supposed to make things easier, faster, or more automatic, that is, less work for the human host. LabVIEW is a unique programming system that makes computer automation a breeze for the scientist or engineer working in many areas of laboratory research, industrial control, and data analysis. You have a job to do—someone is probably paying you to make things happen—and LabVIEW can be a real help in getting that job done, provided that you apply it properly. But debates are now raging over this whole business of computers and their influence on our productivity. For instance, an article I read recently reported that we now tend to write longer proposals and reports (and certainly prettier ones) than we used to when only a typewriter was available. The modern word processor makes it easy to be thorough and communicate ideas effectively. But does this modern approach always result in an improvement in productivity or quality? Sometimes we actually spend *more* time to do the same old thing. We also become slaves to our computers, always fussing over the setup, installing new (necessary?) software upgrades, and generally wasting time.

You must avoid this trap. The key is to analyze your problems and see where LabVIEW and specialized computer hardware can be used to their greatest advantage. Then make efficient use of existing LabVIEW solutions. As you will see, many laboratory automation problems have already been solved for you, and the programs and equipment are readily available. There are no great mysteries here, just some straightforward engineering decisions you have to make regarding the advantages and disadvantages of computer automation. Let's take a pragmatic view of the situation. There are many operations that beg for automation. Among them are

- Long-term, low-speed operations such as environmental monitoring and control
- High-speed operations such as pulsed power diagnostics in which a great deal of data is collected in a short time
- Repetitive operations, such as automated testing and calibration, and experiments that are run many times
- Remote or hazardous operations where it is impractical, impossible, or dangerous to have a human operator present
- High-precision operations that are beyond human capability
- Complex operations with many inputs and outputs

In all these cases, please observe that a computer-automated system makes practical an operation or experiment that you might not otherwise attempt. An automation may offer additional advantages:

- It reduces data transcription errors. The old “graphite data acquisition system” (a pencil) is prone to many error sources not present in a computer data acquisition system. Indeed, more reliable data often leads to better quality control of products and new discoveries in experimental situations.
- It eliminates operator-induced variations in the process or data collection methods. Repeatability is drastically improved because the computer never gets tired and always does things the same way.
- It increases data throughput because you can operate a system at computer speed rather than human speed.

There are some disadvantages hiding in this process of computer automation, however:

- Automation may introduce new sources of error, through improper use of sensors, signal conditioning, and data conversion, and occasionally through computational (for example, round-off) errors.
- Misapplication of any hardware or software system is a ticket for trouble. For instance, attempting to collect data at excessively high rates results in data recording errors.
- Reliability is always a question with computer systems. System failures (crashes) and software bugs plague every high-tech installation known, and they will plague yours as well.

Always consider the cost-effectiveness of a potential automation solution. It seems as if everything these days is driven by money. If you can do it cheaper-better-faster, it is likely to be accepted by the owner, the shareholders, or whoever pays the bills. But is a computer guaranteed to save you money or time? If I have a one-time experiment in which I can adequately record the data on a single strip-chart recorder, an oscilloscope, or with my pencil, then taking two days to write a special program makes no sense whatsoever.

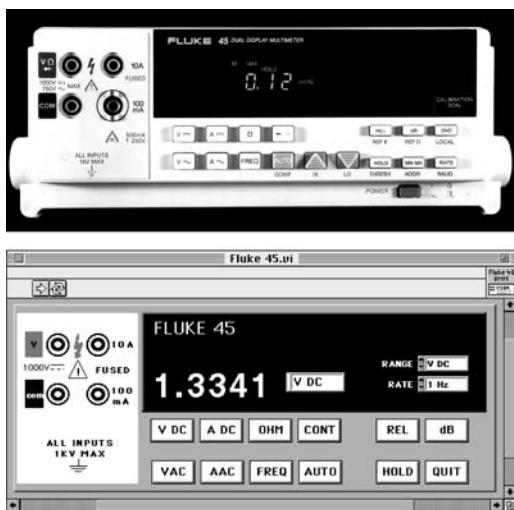
One way to automate (or at least computerize) simple, one-time experiments is to build what I call a LabVIEW *crash cart*, which is much like the doctor’s crash cart in an emergency room. When someone has a short-term measurement problem, I can roll in my portable rack of equipment. It contains a Macintosh with LabVIEW, analog interface hardware, and some programmable instruments. I can quickly configure the general-purpose data acquisition program, record data, and analyze them—all within a few hours. You might want to consider

this concept if you work in an area that has a need for versatile data acquisition. Use whatever spare equipment you may have, recycle some tried-and-true LabVIEW programs, and pile them on a cart. It doesn't even matter what kind of computer you have since LabVIEW runs on all versions of Windows, Power Macintosh, Sun SPARCstations, Linux, and HP workstations. The crash cart concept is simple and marvelously effective.

Automation is expensive: The costs of sensors, computers, software, and the programmer's effort quickly add up. But in the end, a marvelous new capability can arise. The researcher is suddenly freed from the labor of logging and interpreting data. The operator no longer has to orchestrate so many critical adjustments. And data quality and product quality rise. If your situation fits the basic requirements for which automation is appropriate, then by all means consider LabVIEW as a solution.

### **Virtual instruments: LabVIEW's foundation**

LabVIEW made the concept of **virtual instrument (VI)** a practical reality. The objective in virtual instrumentation is to use a general-purpose computer to mimic real instruments with their dedicated controls and displays, but with the added versatility that comes with software. (See Figure 1.1.) Instead of buying a strip-chart recorder, an oscilloscope, and a spectrum analyzer, you can buy one high-performance



**Figure 1.1** This virtual instrument (bottom) is a customized version of the real instrument, but having only the features that you need.

analog-to-digital converter (ADC) and use a computer running LabVIEW to simulate all these instruments and more. The VI concept is so fundamental to the way that LabVIEW works that the programs you write in LabVIEW are in fact called **VI**s. You use simple instruments (**subVI**s) to build more complex instruments, just as you use subprograms to build a more complex main program in a conventional programming language.

**Virtual versus real instrumentation.** Virtual instrumentation systems such as LabVIEW inevitably invite comparison to real physical instrumentation. The major drawback in using a personal computer for implementing virtual instruments is that the computer has only one central microprocessor. An application that uses multiple instruments can easily overburden the processor. A stand-alone instrument, however, may contain any number of processors, each dedicated to specific processing tasks. In addition, these multiple processors can operate in parallel, providing a great increase in overall performance. But this increase in performance results in the expensive price tag accompanying many dedicated instruments, as well as a decrease in flexibility.

Technology from National Instruments is advancing to address these issues, however. First, a new version of LabVIEW—LabVIEW RT—allows you to run independent programs on multiple, dedicated, *real-time* processors that serve to unload tasks from the desktop host. Second, LabVIEW permits you to link all kinds of computing platforms as well as other software applications over a variety of networks, thus further distributing the workload. Third, plug-in boards now contain their own processors and are available at very reasonable prices. Digital signal processors are a good example of special-purpose processors that find their way onto plug-in boards. Many plug-in data acquisition boards also have sophisticated direct memory access (DMA), timing, and triggering capabilities that can span multiple boards, resulting in improved synchronization and signal coupling between boards. These developments, along with the development of more capable operating systems and computer architectures, have brought parallel processing capabilities to personal computers, making them more sophisticated platforms for instrumentation and data acquisition applications. However, sophistication comes at the expense of complexity, because you must have greater knowledge of these hardware components and their interconnection than is required to use a stand-alone instrument with similar capabilities. Virtual instrumentation software is essential for turning these sophisticated hardware combinations into usable instrumentation systems.

Virtual instrumentation offers the greatest benefit over real instruments in the areas of price/performance, flexibility, and customization.

For the price of a dedicated high-performance instrument, you can assemble a personal computer-based system with the fundamental hardware and software components to design virtual instruments targeted for specific applications. The hardware may be plug-in boards, external instruments, or a combination of both. In either case, a software interface can be as complicated or as simple as needed to serve the application. You can simplify the operation of a complex stand-alone instrument with virtual instruments that focus on controlling only subsets of the instrument's full capabilities. I, for one, get lost in the buttons and menus on the panels of many modern instruments and find a simple VI a welcome relief.

Although LabVIEW has existed since 1986, the virtual instrumentation and block diagram concepts embodied in its design are still at the leading edge of instrumentation and computer science technology today. The cost of developing test program software continues to rise with the increasing complexity of the devices being tested and the instruments needed to test them. Software modularity, maintainability, and reusability, key benefits of LabVIEW's hierarchical and homogeneous structure, are critically important to reducing the burden of individual software developers. Reusing routines that you have written and sharing them with others can save a great deal of time and make programs more reliable.

Virtual instrumentation is becoming increasingly important in the instrument control world. **VMEbus Extensions for Instrumentation (VXI)**, a major development in instrumentation, is a standard that defines physical and electrical parameters, as well as software protocols, for implementing instrument-on-a-card test systems. A *VXI instrument* is a card that plugs into a chassis containing several cards. Because they are plug-in cards and not stand-alone boxes, individual VXI instruments do not have front panel user interfaces. Users cannot interact with a VXI instrument by pressing buttons or reading displays on a front panel. Instead, VXI systems must be controlled by a computer or other processor-based device, placing new demands on computer software for controlling instrumentation.

VXI instruments are natural candidates for virtual instrumentation implementations. In the area of user interaction, software front panels offer a visual means of controlling a faceless VXI instrument. In addition, the combination of plug-in modules and high-performance timing and communications capabilities of the VXI bus makes the configuration of a VXI test system much more complex than that of a GPIB test system. LabVIEW's method of graphically connecting virtual instruments into a block diagram accelerates the configuring and programming of VXI test systems.

## Why use LabVIEW?

I use LabVIEW because it has significant advantages over conventional languages and other control and data acquisition packages.

- My productivity is simply better in LabVIEW than with conventional programming languages. I've measured a factor of 5 compared with C on a small project. Others have reported improvements of 15 times.\* Quick prototypes as well as finished systems are now routinely delivered in what used to be record time.
- The graphical-user interface is built in, intuitive in operation, simple to apply, and, as a bonus, nice to look at.
- LabVIEW's graphical language—G—is a *real* programming language, not a specialized application. It has few intrinsic limitations.
- There is only a minimal performance penalty when it is compared with conventional programming languages, and LabVIEW does some things better. No other graphical programming system can make this claim.
- Programmer frustration is reduced because annoying syntax errors are eliminated. Have you ever gotten yourself into an argument with a C compiler over what is considered “legal”? Or made a seemingly minor error with a pointer and had your machine crash?
- Rapid prototyping is encouraged by the LabVIEW environment, leading to quick and effective demonstrations that can be reused or expanded into complete applications.
- Sophisticated built-in libraries and add-on toolkits address specific needs in all areas of science, engineering, and mathematics.
- Programming in LabVIEW is fun. I would *never* say that about C (challenging, yes, but fun, no).

Like any other tool, LabVIEW is useful only if you know how to use it. And the more skilled you are in the use of that tool, the more you will use it. After 19 years of practice, I'm *really comfortable* with LabVIEW. It's gotten to the point where it is at least as important as a word processor, a multimeter, or a desktop calculator in my daily work as an engineer.

---

\*“Telemetry Monitoring and Display Using LabVIEW,” by George Wells, Member of Technical Staff, and Edmund C. Baroth, Manager of the Measurement Technology Center, Jet Propulsion Laboratory, California Institute of Technology. Paper given at the National Instruments User Symposium, March 1993, Austin, Tex.

## The Origin of LabVIEW

A computer scientist friend of mine relates this pseudobiblical history of the computer programming world:

In the beginning, there was only machine language, and all was darkness. But soon, assembly language was invented, and there was a glimmer of light in the Programming World. Then came Fortran, *and the light went out.*

This verse conveys the feeling that traditional computer languages, even high-level languages, leave much to be desired. You spend a lot of time learning all kinds of syntactical subtleties, metaphors, compiler and linker commands, and so forth, just to say “Hello, world.” And heaven forbid that you should want to draw a graph or make something move across the screen or send a message to another computer. We’re talking about many days or weeks of work here. It’s no wonder that it took so many years to make the computer a useful servant of the common person. Indeed, even now it requires at least a moderate education and plenty of experience to do anything significant in computer programming. For the working scientist or engineer, these classical battles with programming languages have been most counterproductive. All you wanted to do is make the darned thing display a temperature measurement from your experiment, and what did you get?

(beep) SYNTAX ERROR AT LINE 1326

Thanks a lot, oh mighty compiler. Well, times have changed in a big way because LabVIEW has arrived. At last, we and other working troops have a programming language that eliminates that arcane syntax, hides the compiler, and builds the graphical-user interface right in. There is no fooling around; just wire up some icons and run. And yet, the thought of actually *programming with pictures* is so incredible when contrasted with ordinary computer languages. How did they do it? Who came up with this idea?

Here is a most enlightening story of the origins of LabVIEW. It’s a saga of vision; of fear and loathing in the cruel world of computer programming; of hard work and long hours; and of breakthroughs, invention, and ultimate success. The original story, *An Instrument That Isn’t Really*, was written by Michael Santori of National Instruments and has been updated for this book.\*

---

\*© 1990 IEEE. Reprinted, with permission, from *IEEE Spectrum*, vol. 27, no. 8, pp. 36–39, August 1990.

## Introduction

Prior to the introduction of personal computers in the early 1980s, nearly all laboratories using programmable instrumentation controlled their test systems using dedicated instrument controllers. These expensive, single-purpose controllers had integral communication ports for controlling instrumentation using the *IEEE-488 bus*, also known as the *General Purpose Interface Bus (GPIB)*. With the arrival of personal computers, however, engineers and scientists began looking for a way to use these cost-effective, general-purpose computers to control benchtop instruments. This development fueled the growth of National Instruments, which by 1983 was the dominant supplier of GPIB hardware interfaces for personal computers (as well as for minicomputers and other machines not dedicated solely to controlling instruments).

So, by 1983, GPIB was firmly established as the practical mechanism for electrically connecting instruments to computers. Except for dealing with some differing interpretations of the IEEE-488 specification by instrument manufacturers, users had few problems physically configuring their systems. The software to control the instruments, however, was not in such a good state. Almost 100 percent of all instrument control programs developed at this time were written in the BASIC programming language because BASIC was the dominant language used on the large installed base of dedicated instrument controllers. Although BASIC had advantages (including a simple and readable command set and interactive capabilities), it had one fundamental problem: like any other text-based programming language, it required engineers, scientists, and technicians who used the instruments to become programmers. These users had to translate their knowledge of applications and instruments into the lines of text required to produce a test program. This process, more often than not, proved to be a cumbersome and tedious chore, especially for those with little or no prior programming experience.

## A vision emerges

National Instruments, which had its own team of programmers struggling to develop BASIC programs to control instrumentation, was sensitive to the burden that instrumentation programming placed on engineers and scientists. A new tool for developing instrumentation software programs was clearly needed. But what form would it take? Dr. Jim Truchard and Jeff Kodosky, two of the founders of National Instruments, along with Jack MacCrisken, who was then a consultant, began the task of inventing this tool. (See Figure 1.2.) Truchard was in search of a software tool that would markedly change the way engineers and scientists approached their test development needs. A model software product that came to mind was the electronic spreadsheet. The spreadsheet addressed the same general problem Truchard, Kodosky, and MacCrisken faced—making the computer accessible to nonprogrammer computer users. Whereas the spreadsheet addressed the needs of financial planners, this entrepreneurial trio



**Figure 1.2** (Left to right): Jack MacCrisken, Jeff Kodosky, and Jim Truchard, LabVIEW inventors.

wanted to help engineers and scientists. They had their rallying cry—they would invent a software tool that had the same impact for scientists and engineers that the spreadsheet had on the financial community.

In 1984, the company, still relatively small in terms of revenue, decided to embark on a journey that would ultimately take several years. Truchard committed research and development funding to this phantom product and named Kodosky as the person to make it materialize. MacCrisken proved to be the catalyst—an amplifier for innovation on the part of Kodosky—while Truchard served as the facilitator and primary user. Dr. T, as he is affectionately known at National Instruments, has a knack for knowing when the product is *right*.

Kodosky wanted to move to an office away from the rest of the company, so he could get away from the day-to-day distractions of the office and create an environment ripe for inspiration and innovation. He also wanted a site close to the University of Texas at Austin, so he could access the many resources available at UT, including libraries for research purposes and, later, student programmers to staff his project. There were two offices available in the desired vicinity. One office was on the ground floor with floor-to-ceiling windows overlooking the pool at an apartment complex. The other office was on the second floor of the building and had no windows at all. He chose the latter. It would prove to be a fortuitous decision.

## All the world's an instrument

The first fundamental concept behind LabVIEW was rooted in a large test system that Truchard and Kodosky had worked on at the Applied Research Laboratory in the late 1970s. Shipyard technicians used this system to test Navy sonar transducers. However, engineers and researchers also had access to the system for conducting underwater acoustics experiments. The system was flexible because Kodosky incorporated several levels of user interaction into its design. A technician could operate the system and run specific test procedures with predefined limits on parameters while an acoustics engineer had access to the lower-level facilities for actually designing the test procedures. The most flexibility was given to the researchers, who had access to all the programmable hardware in the system to configure as they desired (they could also blow up the equipment if they weren't careful). Two major drawbacks to the system were that it was an incredible investment in programming time—over 18 work-years—and that users had to understand the complicated mnemonics in menus in order to change anything.

Over several years, Kodosky refined the concept of this test system to the notion of instrumentation software as a hierarchy of virtual instruments. A virtual instrument (VI) would be composed of lower level virtual instruments, much like a real instrument was composed of printed circuit boards and boards composed of integrated circuits (ICs). The bottom-level VIs represented the most fundamental software building blocks: computational and input/output (I/O) operations. Kodosky gave particular emphasis to the interconnection and nesting of multiple software layers. Specifically, he envisioned VIs as having the same type of construction at all levels. In the hardware domain, the techniques for assembling ICs into boards are dramatically different than assembling boards into a chassis. In the software domain, assembling statements into subroutines differs from assembling subroutines into programs, and these activities differ greatly from assembling concurrent programs into systems. The VI model of homogeneous structure and interface, at all levels, greatly simplifies the construction of software—a necessary achievement for improving design productivity. From a practical point of view, it was essential that VIs have a superset of the properties of the analogous software components they were replacing. Thus, LabVIEW had to have the computational ability of a programming language and the parallelism of concurrent programs.

Another major design characteristic of the virtual instrument model was that each VI had a user interface component. Using traditional programming approaches, even a simple command line user interface for a typical test program was a complex maze of input and output statements often added after the core of the program was written. With a VI, the user interface was an integral part of the software model. An engineer could interact with any VI at any level in the system simply by opening the VI's user interface. The user interface would make it easy to test software modules incrementally and interactively during system development.

In addition, because the user interface was an integral part of every VI, it was always available for troubleshooting a system when a fault occurred. (The virtual instrument concept was so central to LabVIEW's incarnation that it eventually became embodied in the name of the product. Although Kodosky's initial concerns did not extend to the naming of the product, much thought would ultimately go into the name LabVIEW, which is an acronym for *Laboratory Virtual Instrument Engineering Workbench*.)

### A hard-core UNIX guy won over by the Macintosh

The next fundamental concept of LabVIEW was more of a breakthrough than a slow evolution over time. Kodosky had never been interested in personal computers because they didn't have megabytes of memory and disk storage, and they didn't run UNIX. About the time Kodosky started his research on LabVIEW, however, his brother-in-law introduced him to the new Apple Macintosh personal computer. Kodosky's recollection of the incident was that "after playing with MacPaint for over three hours, I realized it was time to leave and I hadn't even said hello to my sister." He promptly bought his own Macintosh. After playing with the Macintosh, Kodosky came to the conclusion that the most intuitive user interface for a VI would be a facsimile of a real instrument front panel. (The Macintosh was a revelation because DOS and UNIX systems in 1983 did not have the requisite graphical user interface.) Most engineers learn about an instrument by studying its front panel and experimenting with it. With its mouse, menus, scroll bars, and icons, the Macintosh proved that the right interface would also allow someone to learn software by experimentation. VIs with graphical front panels that could be operated using the mouse would be simple to operate. A user could discover how they work, minimizing documentation requirements (although people rarely documented their BASIC programs anyway).

### Putting it all together with pictures

The final conceptual piece of LabVIEW was the programming technique. A VI with an easy-to-use graphical front panel programmed in BASIC or C would simplify operation, but would make the development of a VI more difficult. The code necessary to construct and operate a graphical panel is considerably more complex than that required to communicate with an instrument.

To begin addressing the programming problem, Kodosky went back to his model software product—the spreadsheet. Spreadsheet programs are so successful because they display data and programs as rows and columns of numbers and formulas. The presentation is simple and familiar to businesspeople. What do engineers do when they design a system? They draw a block diagram. Block diagrams help an engineer visualize the problem but only suggest a design. Translation of a block diagram to a schematic or computer

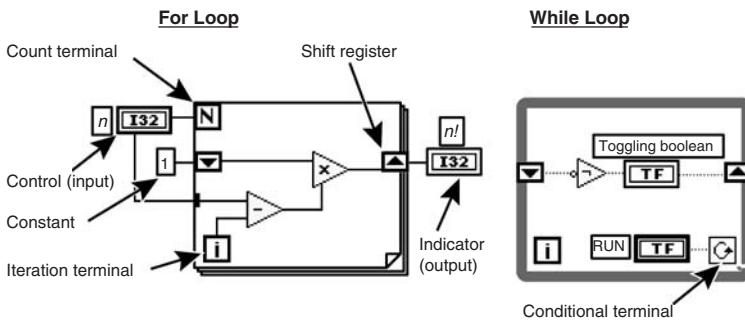
program, however, requires a great deal of skill. What Kodosky wanted was a software-diagramming technique that would be easy to use for conceptualizing a system, yet flexible and powerful enough to actually serve as a programming language for developing instrumentation software.

Two visual tools Kodosky considered were flowcharts and state diagrams. It was obvious that flowcharts could not help. These charts offered a visualization of a process, but to really understand them you have to read the fine print in the boxes on the chart. Thus, the chart occupies too much space relative to the fine print yet adds very little information to a well-formatted program. The other option, a state diagram, is flexible and powerful but the perspective is very different from that of a block diagram. Representing a system as a collection of state diagrams requires a great amount of skill. Even after completion, the diagrams must be augmented with textual descriptions of the transitions and actions before they can be understood.

Another approach Kodosky considered was *dataflow diagrams*. Dataflow diagrams, long recommended as a top-level software design tool, have much in common with engineering block diagrams. Their one major weakness is the difficulty involved in making them powerful enough to represent iterative and conditional computations. Special nodes and feedback cycles have to be introduced into the diagram to represent these computations, making it extremely difficult to design or even understand a dataflow diagram for anything but the simplest computations. Kodosky felt strongly, however, that dataflow had some potential for his new software system.

By the end of 1984, Kodosky had experimented with most of the diagramming techniques, but they were all lacking in some way. Dataflow diagrams were the easiest to work with up until the point where loops were needed. Considering a typical test scenario, however, such as “take 10 measurements and average them,” it’s obvious that loops and iteration are at the heart of most instrumentation applications. In desperation, Kodosky began to make ad hoc sketches to depict loops specifically for these types of operations. *Loops* are basic building blocks of modern structured programming languages, but it was not clear how or if they could be drawn in a dataflow concept. The answer that emerged was a box; a box in a dataflow diagram could represent a loop. From the outside, the box would behave as any other node in the diagram, but inside it would contain another diagram, a *subdiagram*, representing the contents of the loop. All the semantics of the loop behavior could be encapsulated in the border of the box. In fact, all the common structures of structured programming languages could be represented by different types of boxes. His *structured dataflow* diagrams were inherently parallel because they were based on dataflow. In 1990, the first two U.S. patents were issued, covering structured dataflow diagrams and virtual instrument panels. (See Figure 1.3.)

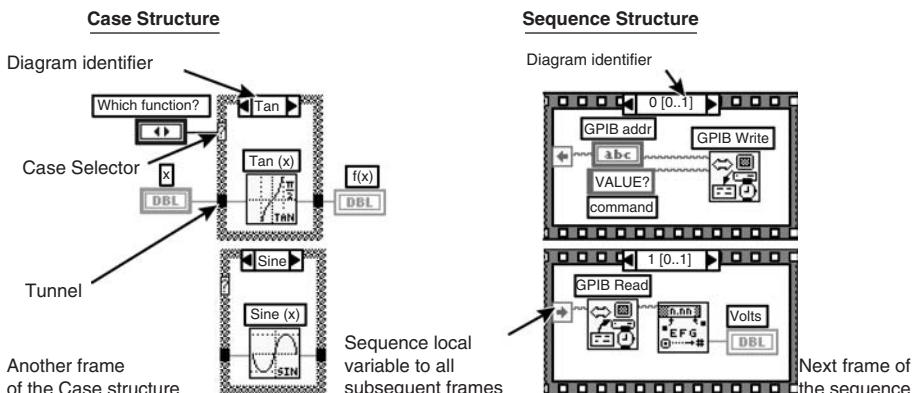
Kodosky was convinced he had achieved a major breakthrough but was still troubled by a nagging point. There are times when it is important to



**Figure 1.3** LabVIEW For Loop and While Loop programming structures with Shift Registers to recirculate data from previous iterations.

force operations to take place sequentially—even when there is no data-flow requiring it. For example, a signal generator must provide a stimulus before a voltmeter can measure a response, even though there isn't an explicit data dependency between the instruments. A special box to represent sequential operations, however, would be cumbersome and take up extra space. During one of their design discussions, Truchard suggested that steps in a sequence were like frames in a movie. This comment led to the notion of having several sequential subdiagrams share the same screen space. It also led to the distinctive graphic style of the **Sequence Structure**, as shown in Figure 1.4.

After inventing the fundamentals of LabVIEW block diagramming, it was a simple matter of using MacPaint to produce pictures of VI panels and diagrams for several common applications. When Kodosky showed them to some engineers at NI, the impact was dramatic. The engineers understood the meaning of the diagrams and correctly guessed how to operate



**Figure 1.4** LabVIEW Case Structures are used for branching or decision operations, and Sequence Structures are used for explicit flow control.

the front panel. Of equal importance, the reviewers expressed great confidence that they would be able to easily construct such diagrams to do other applications. Now, the only remaining task was to write the software, facetiously known as *SMOP: small matter of programming*.

### Favoring the underdog platform for system design

Although Kodosky felt that the graphics tools on the Macintosh made it the computer of choice for developing LabVIEW, the clearer marketing choice was the DOS-based IBM PC. The Macintosh could never be the final platform for the product because it wasn't an open machine, and salespeople would never be able to sell it because the Macintosh was considered a toy by many scientists and engineers. Politics and marketing aside, it wasn't at all clear that you could build a system in which a user draws a picture and the system *runs* it. Even if such a system could be built, would it be fast enough to be useful? Putting marketing concerns aside momentarily, Kodosky decided to build a prototype on the Macintosh prior to building the *real* system on a PC.

Kodosky's affinity for the Macintosh was not for aesthetic reasons alone. The Macintosh system ROM contains high-performance graphics routines collectively known as QuickDraw functions. The Macintosh's most significant graphics capability is its ability to manipulate arbitrarily shaped regions quickly. This capability makes animated, interactive graphics possible. The graphics region algebra performed by the Macintosh is fast because of the unique coordinate system built into QuickDraw: the pixels are between, not on, the gridlines. In addition, the graphics display of the Macintosh uses square pixels, which simplifies drawing in general and rotations of bitmaps in particular. This latter capability proves especially useful for displaying rotating knobs and indicators on a VI front panel.

The operating system of the Macintosh is well integrated. It contains graphics, event management, input/output, memory management, resource and file management, and more—all tuned to the hardware environment for fast and efficient operation. Also, the Macintosh uses Motorola's 68000 family of microprocessors. These processors are an excellent base for large applications because they have large uniform address space (handling large arrays of data is easy) and a uniform instruction set (compiler-generated code is efficient). Remember that this was 1985: the IBM PC compatibles were still battling to break the 640-kilobyte barrier and had no intrinsic graphics support. It wasn't until Microsoft released Windows 3.0 in 1991 that a version of LabVIEW for the PC became feasible.

### Ramping up development

Kodosky hired several people just out of school (and some part-time people still in school) to staff the development team. Without much experience,

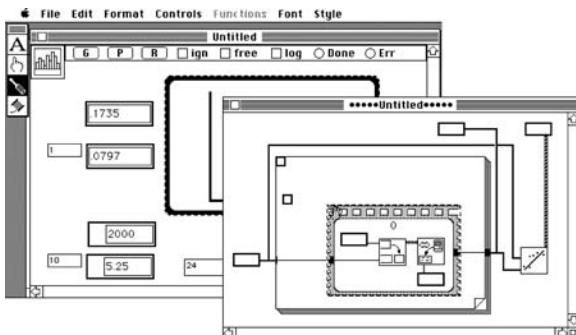
none of the team members was daunted by the size and complexity of the software project they were undertaking and instead they jumped into it with enthusiasm. The team bought 10 Macintoshes equipped with 512 kilobytes of memory and internal hard-disk drives called *HyperDrives*. They connected all the computers to a large temperamental disk server. The team took up residence in the same office near campus used by Kodosky for his brainstorming session. The choice of location resulted in 11 people crammed into 800 square feet. As it turned out, the working conditions were almost ideal for the project. There were occasional distractions with that many people in one room but the level of communication was tremendous. When a discussion erupted between two team members, it would invariably have some impact on another aspect of the system they were inventing. The other members working on aspects of the project affected by the proposed change would enter the discussion and quickly resolve the issue. The lack of windows and a clock also helped the team stay focused. (As it turned out, the developers were so impressed with the productivity of the one-room team concept that the LabVIEW group is still located in one large room, although it now has windows with a great view.)

They worked for long hours and couldn't afford to worry about the time. All-nighters were the rule rather than the exception and lunch break often didn't happen until 3 P.M. There was a refrigerator and a microwave in the room so the team could eat and work at the same time. The main nutritional staples during development were double-stuff Oreo cookies and popcorn, and an occasional mass exodus to Armin's for Middle Eastern food.

The early development proceeded at an incredible pace. In four months' time, Kodosky had put together a team and the team had learned how to program the Macintosh. MacCrisken contributed his project management skills and devised crucial data structure and software entity relationship diagrams that served as an overall road map for software development. They soon produced a proof-of-concept prototype that could control a GPIB instrument (through a serial port adapter), take multiple measurements, and display the average of the measurements. In proving the concept, however, it also became clear that there was a severe problem with the software speed. It would take two more development iterations and a year, before the team would produce a viable product. (See Figure 1.5.)

### Stretching the limits of tools and machine

After finishing the first prototype, Kodosky decided to continue working on the Macintosh because he felt the team still had much to learn before they were ready to begin the real product. It was at this time that the team began encountering the realities of developing such a large software system. The first problems they encountered were in the development tools. The software overflowed some of the internal tables, first in the C compiler and then in the linker. The team worked with the vendor to remedy



**Figure 1.5** Screen shots from a very early prototype of LabVIEW. (Thanks to Paul Daley of LLNL who discovered these old LabVIEW versions deep in his diskette archive.)

the problem. Each time it occurred, the vendor expanded the tables. These fixes would last for a couple months until the project grew to overflow them again. The project continued to challenge the capabilities of the development tools for the duration of the project.

The next obstacle encountered was the Macintosh jump table. The project made heavy use of object-oriented techniques, which resulted in lots of functions, causing the jump tables to overflow. The only solution was to compromise on design principles and work within the limits imposed by the platform. As it turned out, such compromises would become more commonplace in the pursuit of acceptable performance.

The last major obstacle was memory. The project was already getting too large for the 512-kilobyte capacity of Macintosh and the team still hadn't implemented all the required functions, let alone the desirable ones they had been hoping to include. The prospects looked dim for implementing the complete system on a DOS-based PC, even with extensive use of overlaying techniques. This situation almost proved fatal to the project. The team was at a dead end and morale was at an all-time low. It was at this opportune time that Apple came to the rescue by introducing the Macintosh Plus in January 1986. The Macintosh Plus was essentially identical to the existing Macintosh except that it had a memory capacity of one megabyte. Suddenly, there was enough memory to implement and run the product with most of the features the team wanted.

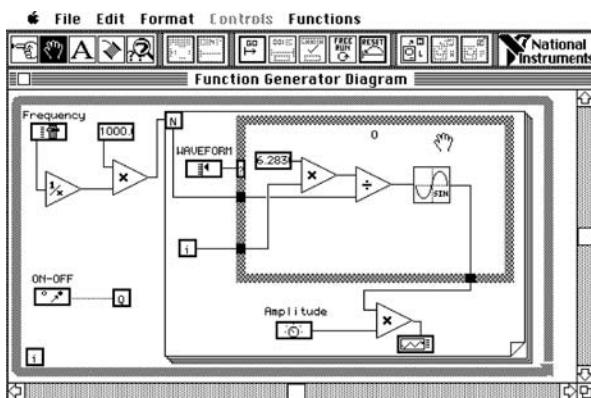
Once again, the issue of the marketability of the Macintosh arose. A quick perusal of the DOS-based PC market showed that the software and hardware technology had not advanced very much. Kodosky decided (with approval by Dr. Truchard after some persuasion) that, having come this far on the Macintosh, they would go ahead and build the first version of LabVIEW on the Macintosh. By the time the first version of LabVIEW was complete, there would surely, they thought, be a new PC that could run large programs.

### Facing reality on estimated development times

The initial estimates of the remaining development effort were grossly inaccurate. The April 1986 introduction date passed without a formal software release. In May, in anticipation of an imminent shipping date, the team moved from their campus workroom to the main office, where they could be close to the application engineers who did customer support. This event caused much excitement but still no product.

It was at this point that the company became over-anxious and tried to force the issue by prematurely starting beta testing. The testing was a fiasco. The software was far from complete. There were many bugs encountered in doing even the most simple and common operations. Development nearly ground to a halt as the developers spent their time listening to beta testers calling in the same problems. (See Figure 1.6.)

As the overall design neared completion, the team began focusing more on details, especially performance. One of the original design goals was to match the performance of interpreted BASIC. It was not at all clear how much invention or redesign it would require to achieve this performance target, making it impossible to predict when the team would achieve this goal. On most computational benchmarks, the software was competitive with BASIC. There was one particular benchmark, the Sieve of Eratosthenes, that posed, by nature of its algorithm and design, particular problems for dataflow implementations. The performance numbers the team measured for the sieve benchmark were particularly horrendous and discouraging—a fraction of a second for a compiled C program, two minutes for interpreted BASIC, and over eight hours for LabVIEW.



**Figure 1.6** This diagram screen shot is from LabVIEW beta 0.36 in June 1986. The familiar structures (While and For Loops, and a Case Structure) had appeared by this time.

Kodosky did his best to predict when the software would be complete, based on the number of bugs, but was not sure how the major bugs would be found, much less fixed. Efficiently testing such a complex interactive program was a vexing and complex problem. The team finally settled on the *bug day* approach. They picked a day when the entire team would stop working on the source code and simply use LabVIEW. They would try all types of editing operations, build as many and varied VIs as they could, and write down all the problems they encountered until the white boards on every wall were full. The first bug day lasted only three hours. The team sorted the list and for the next five weeks fixed all the fatal flaws and as many minor flaws as possible. Then they had another bug day. They repeated this process until they couldn't generate even one fatal flaw during an entire day. The product wasn't perfect, but at least it would not be an embarrassment.

### **Shipping the first version**

In October 1986, the team figured out how to bypass some of the overhead in calling a subVI, producing some improvement in performance (for all but the sieve benchmark, which was better, but still 20 times slower than BASIC). The decision was made to ship the product. The team personally duplicated and packaged the first 50 disks and hand-carried them to shipping. Version 1.0 was on the streets.

The reaction to LabVIEW was, in a word, startling. The product received worldwide acclaim as the first viable, *visual*, or graphical language. There were many compliments for a well-designed product, especially from research and development groups who had had their Macintosh-based projects canceled by less-adventurous CEOs and marketing departments. Interestingly enough, the anticipated demand of the targeted BASIC users did not materialize. These people were apparently content to continue programming as they had been doing. Instead, LabVIEW was attracting and eliciting demands from customers who had never programmed at all but who were trying to develop systems considered extremely difficult by experienced programmers in any language. Yet these customers believed they could successfully accomplish their application goals with LabVIEW.

### **Apple catches up with the potential offered by LabVIEW**

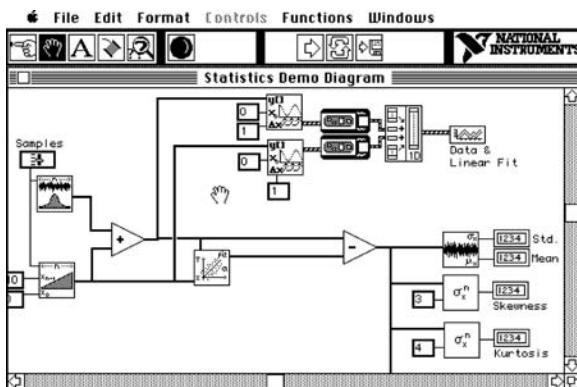
Shortly after shipment of LabVIEW began, the company received its first prototype of the Macintosh II. This new version of the Macintosh had many design features that promised to legitimize the Macintosh in the scientific and engineering community. The most important of these features was the open architecture of the new machine. Previous Macintosh versions did not have the capability to accept plug-in expansion boards. The only mechanisms available for I/O were RS-422 serial and SCSI (Small Computer Systems Interface) ports. National Instruments sold stand-alone interface

box products that converted these ports to IEEE-488 control ports, but performance suffered greatly.

The Macintosh II's open architecture made it possible to add not only IEEE-488 support but also other much-needed I/O capabilities, such as analog-to-digital conversion and digital I/O. The Macintosh II used the NuBus architecture, an IEEE standard bus that gave the new machine high-performance 32-bit capabilities for instrumentation and data acquisition that were unmatched by any computer short of a minicomputer (the PC's bus was 16 bits). With the flexibility and performance afforded by the new Macintosh, users now had access to the hardware capabilities needed to take full advantage of LabVIEW's virtual instrumentation capabilities. Audrey Harvey (now a system architect at National Instruments) led the hardware development team that produced the first Macintosh II NuBus interface boards, and Lynda Gruggett (now a LabVIEW consultant) wrote the original *LabDriver* VIs that supported this new high-performance I/O. With such impressive new capabilities and little news from the PC world, National Instruments found itself embarking on another iteration of LabVIEW development, still on the Macintosh.

Effective memory management turned out to be the key to making this graphical language competitive with ordinary interpreted languages. The development team had used a literal interpretation of dataflow programming. Each time data (a wire) leaves a source (a node), the Macintosh Memory Manager is called to allocate space for the new data, adding tremendous overhead. Other performance factors involved effective interpretation of the VI and diagram hierarchy and the scheduling of execution among nodes on the diagram. It became obvious that memory reuse was vital, but a suitable algorithm was far from obvious. Kodosky and MacCrisken spent about four intensive weekend brainstorming sessions juggling the various factors, eventually arriving at a vague algorithm that appeared to address everything. The whiteboard was covered with all sorts of instantiation diagrams with "little yellow arrows and blue dots" showing the prescribed flow of data. Then, one Monday morning, they called in Jeff Parker (now a LabVIEW consultant) and Steve Rogers (still a LabVIEW developer at National Instruments) and introduced them to this magnificent algorithm. The two of them proceeded to implement the algorithm (to the point that Kodosky and MacCrisken admittedly don't understand it anymore!). They kept the famous whiteboard around for about a year, occasionally referring to it to make sure everything was right. LabVIEW 1.1 included these concepts, known collectively as *inplaceness*.

While the development team was working with the new Macintosh, they were also scrambling to meet some of the demands made by customers. They were simultaneously making incremental improvements in performance, fixing flaws that came to light after shipment began, and trying to plan future developments. As a result of this process, LabVIEW progressed from version 1.0 to version 1.2 (and the sieve progressed to 23 seconds). LabVIEW 1.2 was a very reliable and robust product. I,



**Figure 1.7** This is LabVIEW 1.2. It ran only in black and white and you couldn't move an object once it was wired. Somehow, we early users managed to get a lot accomplished and enjoyed ourselves, at that.

for one, wrote a lot of useful programs in version 1.2 and can't recall crashing. (See Figures 1.7 and 1.8.)

The Macintosh II gave LabVIEW a much-needed and significant boost in performance. The improvement, however, was short-lived. The internal architecture of LabVIEW 1.2 was showing signs of distress and the software was apparently abusing the Macintosh resource manager as well as its memory manager. It was becoming clear that the only real way to enhance capabilities was with a complete redesign. At the least, a redesign could incorporate new diagram analysis algorithms and a fast built-in compiler that would eliminate performance problems once and for all. The major objective of the redesign was to achieve execution performance within a factor of two of compiled C.



**Figure 1.8** This is how we got to know the LabVIEW 1 development team: the About LabVIEW dialog box had these way-cool portraits.

## LabVIEW 2: A first-rate instrument control product becomes a world-class programming system

Even as the plans for the next-generation LabVIEW were becoming firm, customers were fast approaching, and exceeding, the limits of LabVIEW 1.2. Some users were building systems of VIs using up to 8 MB of memory (then the limit on a Macintosh II). The *mega-applications* took up to 30 minutes to load into memory. Users reeled at the nefarious *Too many objects* error message. And members of the development team often shuddered when they thought of the huge number of allocated structures and the complexity of their interconnection needed to make such a system work.

The decision to redesign LabVIEW brought with it a new set of pressures. In responding to some of the customer demands, the company had to admit that version 1.2 was at its design limits. As a result, work began on a new version, already becoming known as LabVIEW 2. Once the word was out, the development team was pressured into predicting a release date. Despite Kodosky's best intentions, the scope of the redesign resulted in several missed shipping deadlines. He realized that the design of a hierarchical dataflow compiler with polymorphic functions and sophisticated memory management was a new science—and it took awhile.

LabVIEW 2 was designed with formalized object-oriented programming (OOP) techniques, on the insistence of Jeff Parker. OOP has many advantages over common procedural languages, and in fact it was an enabling technology in this case. Unfortunately, OOP tools for C language development were in a rather primitive state in the 1988 time frame, so the team wrote their own spreadsheet-based development tools to automatically generate source code files and to keep track of objects and methods. These tools remain in use because they are more efficient than C++ in terms of code size and performance, though Kodosky reports that C++ is under serious consideration now for reasons of future support and portability.

The development team released an alpha version of the software in late 1988. Cosmetically, this version appeared to be in excellent shape. The compiler was working well and showed great increases in performance. There were also many enhancements in the editing capabilities of the product. All these positive signs resulted in an air of excitement and of imminent release. Unfortunately, the team had a long list of items they knew had to be fixed to produce a technically sound product. Over a year elapsed before the team released the final product. In January 1990, LabVIEW 2 shipped to the first eager customers. I have to say that being a beta tester was a real thrill: the improvement in speed and flexibility was astounding.

LabVIEW 2's compiler is especially notable not only for its performance but for its integration into the development system. Developing in a standard programming language normally requires separate compilation and linking steps to produce an executable program. The LabVIEW 2 compiler is



**Figure 1.9** The team that delivered LabVIEW 2 into the hands of engineers and scientists (clockwise from upper left): Jack Barber, Karen Austin, Henry Velick, Jeff Kodosky, Tom Chamberlain, Deborah Battó, Paul Austin, Wei Tian, Steve Chall, Meg Fletcher, Rob Dye, Steve Rogers, and Brian Powell. Not shown: Jeff Parker, Jack MacCrisken, and Monnie Anderson.

an integral and invisible part of the LabVIEW system, compiling diagrams in a fraction of the time required by standard compilers. From a user's point of view, the compiler is so fast and invisible that LabVIEW 2 is every bit as interactive as the previous interpreted versions. (See Figure 1.9.)

### The port to Windows and Sun

The next major quest in the development of LabVIEW was the portable, or platform-independent version. Dr. Truchard (and thousands of users) had always wanted LabVIEW to run on the PC, but until Windows 3.0 came along, there was little hope of doing so because of the lack of 32-bit addressing support that is vital to the operation of such a large, sophisticated application. UNIX workstations, on the other hand, are well suited to such development, but the workstation market alone was not big enough to warrant the effort required. These reasons made Kodosky somewhat resistant to the whole idea of programming on the PC, but MacCrisken finally convinced him that *portability* itself—the isolation of the machine-dependent layer—is the real challenge. So, *the port* was on.

Microsoft Windows 3 turned out to be a major kludge with regard to 32-bit applications (Windows 3, itself, and DOS are 16-bit applications). Only in Appendix E of the Windows programming guide was there any mention whatsoever of 32-bit techniques. And most of the information contained

in that appendix referred to storage of data and not applications. Finally, only one C compiler—Watcom C—was suited to LabVIEW development. But before Watcom C became available, Steve Rogers created a set of glue routines that translates 32-bit information back and forth to the 16-bit system function calls (in accordance with Appendix E). He managed to successfully debug these low-level routines without so much as a symbolic debugger, living instead with hexadecimal dumps. This gave the development team a six-month head start. Rogers summed up the entire situation: “It’s ugly.” Development on the Sun SPARCstation, in contrast, was a relative breeze. Like all good workstations, the Sun supports a full range of professional development tools with few compromises—a programmer’s dream. However, the X Windows environment that was selected for the LabVIEW graphical interface was totally different from the Macintosh or Windows toolbox environments. A great deal of effort was expended on the low-level graphics routines, but the long-term payoff is in the portability of X Windows-based programs. Development on the Sun was so convenient, in fact, that when a bug was encountered in the Windows version, the programmer would often do his or her debugging on the Sun rather than suffering along on the PC. Kodosky reports, “The Sun port made the PC port much easier and faster.” LabVIEW 2.5, which was released in August 1992, required rewriting about 80 percent of LabVIEW 2 to break out the machine-dependent, or *manager*, layer. Creating this manager layer required some compromises with regard to the look and feel of the particular platforms. For instance, creating *floating windows* (such as the LabVIEW Help window) is trivial on the Macintosh, difficult on the PC, and impossible under X Windows. The result is some degree of least-common-denominator programming, but the situation has continued to improve in later versions through the use of some additional machine-dependent programming. (See Figure 1.10.)

### LabVIEW 3

The LabVIEW 2.5 development effort established a new and flexible architecture that made the unification of all three versions in LabVIEW 3 relatively easy. LabVIEW 3, which shipped in July of 1993, included a number of new features beyond those introduced in version 2.5. Many of these important features were the suggestions of users accumulated over several years. Kodosky and his team, after the long and painful port, finally had the time to do some really creative programming. For instance, there had long been requests for a method by which the characteristics of controls and indicators could be changed programmatically. The *Attribute Node* addressed this need. Similarly, *Local Variables* made it possible to both read from and write to controls and indicators. This is an extension of strict dataflow programming, but it is a convenient way to solve many tricky problems. Many subtle compiler improvements were also made that enhance performance, robustness, and extensibility. Additional U.S. patents were issued in 1994 covering these extensions to structured dataflow diagrams such as globals and locals, occurrences, attribute nodes, execution highlighting, and so forth.



**Figure 1.10** “Some new features are so brilliant that eye protection is recommended when viewing.” The team that delivered LabVIEW 2.5 and 3.0 includes (left to right, rear) Steve Rogers, Thad Engeling, Duncan Hudson, Keving Woram, Greg Richardson, Greg McKaskle; (middle) Dean Luick, Meg Kay, Deborah Batt-Bryant, Paul Austin, Darshan Shah; (seated) Brian Powell, Bruce Mihura. Not pictured: Gregg Fowler, Apostolos Karmirantzos, Ron Stuart, Rob Dye, Jeff Kodosky, Jack MacCrisken, Stepan Riha.

LabVIEW 3 was a grand success, both for National Instruments and for the user community worldwide. Very large applications and systems were assembled containing thousands of VIs. The *LabVIEW Application Builder* permitted the compilation of true stand-alone applications for distribution. Important add-on toolkits were offered by National Instruments and third parties, covering such areas as process control, imaging, and database access. Hundreds of consultants and corporations have made LabVIEW their core competency; speaking as one of them, the phones just keep on ringing.

#### LabVIEW 4

Like many sophisticated software packages, LabVIEW has both benefited and suffered from *feature creep*: the designers respond to every user request, the package bulks up, and pretty soon the beginner is overwhelmed. April 1996 brought LabVIEW 4 to the masses, and, with it, some solutions to perceived ease-of-use issues. Controls, functions, and tools were moved into customizable floating palettes, menus were reorganized, and elaborate online help was added. Tip strips appear whenever you point to icons, buttons, or other objects. Debugging became much more powerful. And even the manuals received an infusion of valuable new information. I heard significant, positive feedback from new users on many of these features.

LabVIEW 4 enabled the assembly of very large applications consisting of thousands of VIs, including stand-alone executables, courtesy of the *LabVIEW Application Builder*. Most important, National Instruments realized that LabVIEW was a viable development platform with regard to creating new features and even extending LabVIEW itself. The first such extension was the *DAQ Channel Wizard*, a fairly sophisticated application that provided a comprehensive user interface for managing channel configurations on plug-in boards. Development time was drastically reduced compared to conventional C-based coding. Audrey Harvey began the Channel Wizard development in LabVIEW, showing key features in prototypical fashion. When it became apparent that her prototype met most of the perceived needs of the user community, it was handed off to Deborah Battobryant, who perfected it for release with LabVIEW 4.0. This strategy has since been followed many times. These LabVIEW-based features, when well designed, are indistinguishable from their C-based counterparts. The underlying plumbing in LabVIEW 4 was not significantly changed from LabVIEW 3, though some advanced R&D activities were certainly under way. The team called the next generation LabVIEW 10, referring to some distant, future release, and split off a special group dedicated to new features such as undo, multithreading, and real-time capability. While some of these features (including a simple form of *undo*) almost made it into LabVIEW 4.1, they had to wait until the next version.

### LabVIEW branches to BridgeVIEW

National Instruments decided to target the industrial process control market—initially with LabVIEW extensions—and ended up launching a new product, **BridgeVIEW**. It's important to make a distinction between LabVIEW—the development environment—and G, the underlying programming language. The two are indeed separable. BridgeVIEW is an independent development environment that uses the G language and compiler, but is optimized for process control applications. Primary differences between the two products include some changes in the menus, the inclusion of a real-time process control database, extensive system configuration tools, a new driver concept, and a reduction in the amount of programming required to create a substantial process control application. The result is a ready-to-use process control package that you can freely extend with G language diagrammatic programming. This contrasts sharply with other process control packages that *do what they do*, and no more. LabVIEW users with a control application will find BridgeVIEW very comfortable.

### LabVIEW 5

For LabVIEW 5, Jeff Kodosky's team adopted completely new development practices based on a formalized, milestone-driven scheme that explicitly measured software quality. (See Figure 1.11.) Bugs were tracked quantitatively during all phases of the development cycle, from code design to user beta testing. The result, in February 1998, was the most reliable LabVIEW release ever, in spite of its increased complexity.



**Figure 1.11** The LabVIEW development milestones.

The quality improvement process began with a thorough scrubbing of LabVIEW 4—a laborious but necessary task—prior to adding new features. Then the fruits of the prior LabVIEW 10 efforts and other advanced concepts were incorporated. Ultimately, a good deal of the application was rewritten.

**Undo** was positively the most-requested feature among users, all the way back to the dawn of the product. But it was also considered one of the most daunting. Behind the scenes, LabVIEW is an extremely complex collection of objects—orders of magnitude more complicated than any word processor or drawing program. The simpleminded approach to implementing undo is simply to duplicate the entire VI status at each editing step. However, this quickly leads to excessive memory usage, making only a single-level undo feasible. Instead, the team devised an incremental undo strategy that is fast, memory efficient, and supports multiple levels. It's so unique that it's patented. Of course, it took a while to develop, and it involved a complete rewrite of the LabVIEW editor. That was step 1.

Step 2 required rewriting the LabVIEW execution system to accommodate another advanced feature: **multithreaded** execution. Applications that support multiple threads use the scheduling capabilities of the computer's operating system to divide CPU time between various tasks, or threads. For instance, the user interface thread can run independently of the data acquisition thread and the data analysis thread, and all of those threads can have different priorities. While the potential performance enhancements are great, multithreading has traditionally been a fairly difficult programming challenge tackled only by skilled programmers. The LabVIEW 5 team brought multithreaded development to the masses by providing a simple, low-risk interface.

As any programmer can tell you, the simpler a feature appears, the more complex is its underlying code—and that certainly describes the conversion of LabVIEW to multithreading. The major task was to evaluate every one of the thousands of functions in LabVIEW to determine whether it was *thread-safe*, *reentrant*, or needed some form of protection. This evaluation process was carried out by first writing scripts in PERL (a string manipulation language) that analyzed every line of code in the execution system, and then having the programmers perform a second (human) evaluation. As it turned out, about 80 percent of the code required no changes, but every bit had to be checked. Kodosky said the process was “... like doing a heart and intestine transplant.”

The third and final step in the Great Rewrite involved the compiler. As you know by now, LabVIEW is unique in the graphical programming world in that it directly generates executable object code for the target machine. Indeed, that compiler technology is one of the crown jewels at National Instruments. At the time, one of the latent shortcomings in the compiler was the difficulty involved in supporting multiple platforms. Every time a new one was added, a great deal of duplicated effort was needed to write another platform-specific compiler process. The solution was to create a new platform-independent layer that consolidated and unified the code prior to object code generation.

Now adding a new platform involves only the lowest level of object code generation, saving time and reducing the chances for new bugs. It also reduced the code size of the compiler by several thousand lines. It's another of those underlying technology improvements that remains invisible to the user except for increased product reliability.

### The LabVIEW RT branch

As long as any of us can remember, we've been asking for enhanced realtime performance from LabVIEW (this topic is covered in detail in Chapter 21, “LabVIEW RT Does Real Time”). It's also been a hot topic of discussion for the LabVIEW team as well, and they finally came up with a solution that was released in the form for LabVIEW RT in May 1999. After years of discussion and conceptual design that hadn't completely gelled into a solution, Jeff Kodosky offered a personal challenge to Darshan Shah: “I bet you can't!” Needless to say, Darshan was up to the challenge.

LabVIEW RT required a number of innovations. First, a specialized realtime operating system (RTOS) was needed to provide faster and more predictable scheduling response than a desktop OS. RTOSs often lack many of the services that standard LabVIEW requires, and they must be configured for a specific hardware configuration. Darshan chose PharLap TNT Embedded because it supported many Windows Win32 API (application programmer interface) calls, which made it much easier to port LabVIEW as well as NI-DAQ. To simplify hardware compatibility problems, the first release of LabVIEW RT ran only on a special DAQ board married to an

80486 processor, which was chosen for its code compatibility and adequate performance at low cost. It was a reasonable engineering decision for an initial product release.

A second challenge was creating a simple and intuitive way of interacting with the embedded LabVIEW RT processor board. Darshan's solution was to provide a simple configuration dialog that tells LabVIEW to compile and run on either the host machine or the embedded board. This completely shields the user from the underlying platform change and all the real-time complexity that one would normally have to face. Another benefit of this architecture was that its development was synergistic with the other new LabVIEW 5 concepts: the division between the user interface and execution systems, and multithreading.

## LabVIEW 6

June 2000 brought us LabVIEW 6, with a host of new user-interface features (rendered in 3D, no less), extensive performance and memory optimization at all levels, and perhaps most important, a very powerful **VI Server**. Introduced in LabVIEW 5, the VI Server gives the user external hooks into VIs. For instance, you can write programs that load and run VIs, access VI documentation, and change VI setup information, all without directly connecting anything to the target VI. Furthermore, most of these actions can be performed by applications other than LabVIEW, including applications running elsewhere on a network. This new paradigm effectively publishes LabVIEW, making it a true member of an application system rather than exclusively running the show as it had in the past. You can even export shared libraries (DLLs) in LabVIEW. From the perspective of professional LabVIEW programmers, we believe that LabVIEW 6 has truly fulfilled the promise of graphical programming.

At the nuts-and-bolts level, LabVIEW 6 is now completely developed in C++, in contrast to all the earlier versions, which used C with object extensions created by the LabVIEW team. They stuck with C for so long because C++ compiler effectiveness was, in their judgment, not as good as C. As a point of reference, LabVIEW 6 consists of about 800,000 lines of code. While that's a pretty big number, there are plenty of applications running on your very own computer that are much heftier (including NI-DAQ, but that's another story). Also, the growth rate in LabVIEW's code size has actually diminished over the years due to improved programming techniques. It's always impressive to see continuous improvement in an application's performance and efficiency, rather than pure bloat as new features are added (see Figure 1.12).

## LabVIEW 7

LabVIEW 7 Express was released in April of 2003 with new features and wizards designed to make programming and getting started easier for



Figure 1.12 My, how we have grown! This is most of the LabVIEW 6 team.

people new to LabVIEW. Express blocks make configuring instruments, testing analysis routines, and most common tasks simple. The DAQ Assistant made data acquisition easier for everyone. Now with a few clicks of a mouse anyone can create a fully functioning data acquisition program. You're not limited to simple tasks either; the DAQ Assistant makes complex trigger applications a snap. The LabVIEW team went out of their way to help the novice with new, simplified ways of doing things without fatally compromising the power of LabVIEW.

LabVIEW 7 also brings LabVIEW to new platforms far from the desktop. Targets include a variety of compact computers and operating systems, including Windows CE, Palm OS, and the ARM processor family. New in LabVIEW is a compiler for **field-programmable gate arrays (FPGAs)**. LabVIEW for FPGA continues National Instruments' push of LabVIEW Everywhere. This is more than just marketing hype. With LabVIEW FPGA, engineers are now able to convert LabVIEW VIs into hardware implementations of G code.

## LabVIEW 8

Released in October 2005, LabVIEW 8 introduced new tools to make LabVIEW developers more productive and application development and integration across a wide range of platforms easier. The LabVIEW project provides a cohesive environment for application developers. Finally, at long last, we can simultaneously develop applications for multiple targets. Developers using LabVIEW FPGA and RT can develop applications on the host and the target without having to close one environment and switch to another. The project interface provides a relatively painless way to manage development and deployment of large applications by many developers. LabVIEW is the number one programming language in test and measurement and it's time we had some big time tools to manage our projects. Look for more object oriented programming as LabVIEW includes built in OOLV tools. National Instruments continues to take LabVIEW deeper into education and simulation with the educational version of LabVIEW for DSPs and the new LabVIEW MathScript. MathScript is an integral part of LabVIEW that combines dataflow with text-based mathematical programming. And as shown in Figure 1.13, LabVIEW keeps growing.



**Figure 1.13** LabVIEW keeps growing and so does the development team. This is the team circa 2003. Photo courtesy of National Instruments Corporation.

## Crystal Ball Department

Jeff Kodosky's predictions in the last edition proved to be amazingly accurate, so we asked him to share some rumors regarding future trends and features for LabVIEW. Here's Jeff:

Probably the most important future goals fall in the area of fleshing out LabVIEW's design capabilities. We have always had a contingent of customers doing system design using LabVIEW but the strongest area for us was virtual instrumentation ("acquire-analyze-present"). As LabVIEW has grown, particularly with the introduction of LabVIEW Real Time and more recently LabVIEW FPGA, it has become much more suitable for hard realtime control applications. But control applications typically are developed in conjunction with modeling or simulating the system under control, so we have been introducing toolkits for system identification and control design and simulation, etc., and all these design capabilities fit superbly well with LabVIEW's graphical dataflow paradigm. There are also other "models of computation" that are important for many types of design and we have several integrated into LabVIEW already (e.g., state diagrams, text math in the formula box, event structure, network variable) and others that we are continuing to develop (e.g., synchronous data flow, the next Lego "Robolab," other specialized models specific to application areas). One might even consider a circuit diagram to be a model of computation, so to that end we are exploring much tighter coupling between LabVIEW and Electronics WorkBench Multisim.

Our current focus is nicely summarized by our new description: design-prototype-deploy, where LabVIEW is ideally suited to all aspects of the process from design, to prototype (in conjunction with platforms like PXI and cRIO), to deployment (on the same platforms for low volumes or custom platforms for high volumes).

In the past, LabVIEW was typically the tool for testing the widgets being produced, but now it is also helping design the widget and may even be part of the widget.

### LabVIEW influences other software products

The concepts found in LabVIEW have influenced the design of many other instrumentation products, especially in the areas of software front panels and iconic programming. Such products exist for Macintosh, DOS, Windows, and UNIX-based computers. Although at first glance these products are easily confused with LabVIEW, the differences are many.

As LabVIEW does, most instrumentation software products now offer some form of front panel-oriented user interface. The role of a LabVIEW front panel is unique, however, in comparison to other products. A graphical front panel in most instrumentation software products is simply a display/user-interface mechanism introduced into an otherwise

conventional, textual language-based programming system. National Instruments' **LabWindows CVI** is such a package. The sole purpose of the panel in such a system is to serve as a top-level interface to an instrument or plug-in board. In some packages, users don't have the flexibility to customize their own front panel; it's merely a user-friendly tool supplied by the vendor. These panels are either not applicable to or not available as an interface for use within an application program.

In LabVIEW, a front panel is an integral component of every software module (or VI). A LabVIEW front panel can serve as the interface to any type of program, whether it is a GPIB instrument driver program, a data acquisition program, or simply a VI that performs computations and displays graphs. The methodology of using and building front panels is fundamental to the design and operation of LabVIEW; for every front panel, there is an associated block diagram program and vice versa. Front panels can appear at any level in a LabVIEW VI and are not restricted to the highest level.

Another aspect of LabVIEW that finds its way into other software products is **iconic programming**. As with front panels, the iconic programming systems found in other products are largely add-on shells used to hide some of the details of systems based on conventional programming methodologies. Such products provide a predefined, set number of operations that can be used in iconic form. While generally quite simple to use because of their turnkey nature, these systems are not readily expanded because adding new functions is either impossible or requires a user to revert back to a standard programming language to create new icons. These iconic programming systems address organization of the high-level aspects of a test, but do not have graphical constructs for specifying programming operations such as execution order, iteration, or branching. In addition, these systems typically do not support building a module that can be used in iconic form at a higher level. Finally, and perhaps most importantly, LabVIEW is a compiled language, whereas all other iconic packages are interpreted. A 100-fold increase in performance is typically demonstrated.

LabVIEW's iconic programming language is not simply an organizational tool. It is a true programming language complete with multiple data types, programming structures, and a sophisticated compiler. LabVIEW is unique for its hierarchy and expandability. VIs have the same construction at all levels, so the creation of a new high-level VI is simply a matter of combining several lower-level VIs and putting a front panel on the new combination. Because LabVIEW's graphical programming system offers the functionality and performance of conventional programming languages, users can implement and add new functions to the system without having to resort to the use of another programming language.

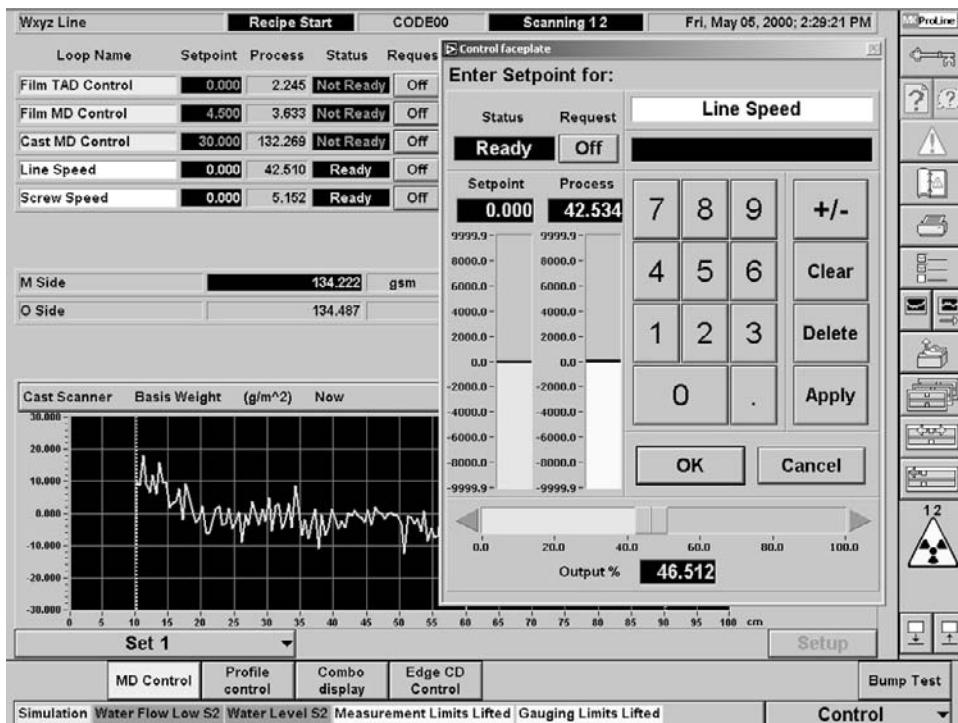
## LabVIEW Handles Big Jobs

Some really impressive applications have been written in all versions of LabVIEW, but in the past few years we've seen an absolute explosion in development that leaves us breathless. Even the LabVIEW team members have been known to shake their heads in amazement when shown the size and scope of the latest high-end project from a sophisticated user. Here are a few benchmark projects that we've run across.

At the **Lawrence Livermore National Laboratory**, Gary was responsible for the LabVIEW software architecture that controlled a *femtosecond laser cutting system*. This unique and very complex system of lasers was developed at a cost of \$20 million for a high-value defense application. Four Windows NT workstations were networked together, with LabVIEW providing measurement, control, and automatic sequencing for the laser itself, as well as a high-level user interface for semiskilled operators. Nearly every type of I/O hardware was installed: DAQ, GPIB, serial, remote SCXI, VME/VXI, and IMAQ imaging. Formal software quality assurance procedures were applied because of the mission-critical nature of the project. A total of about 4 work-years of software development went into the creation of about 600 VIs.

At *Spectrum Astro, Inc.*, Roger Jellum and Tom Arnold developed **AstroRT**, a LabVIEW-based data acquisition and control system for factory test and on-orbit operations of spacecraft. Now marketed as a commercial package, AstroRT has been used for many missions, including the Air Force Research Lab *MightySat II.1* spacecraft, NASA's *New Millennium Deep Space 1*, TDRSS and Swift spacecraft, and the Naval Research Lab's *Coriolis* spacecraft. Running under Windows NT, AstroRT provides the capability to collect, process, and distribute frame-synchronized or packetized telemetry received from a spacecraft or subsystem. The system can also transmit commands to the spacecraft, interface with ground-support equipment (typically via GPIB, VXI, or Ethernet interfaces), display and archive data, and assist with mission planning. The whole package encompasses more than 2000 VIs and requires about 7 work-years to develop.

Perhaps the grandest LabVIEW application of all is **MXProLine**, developed by a team lead by Dirk Demol at **Honeywell-Measurex Corporation**. Measurex provides process control equipment and systems for the production of sheet goods, particularly paper and plastic films. MXProLine is a state-of-the-art distributed process control system with LabVIEW representing about 95 percent of the code base (See Figure 1.14). Dirk's team worked closely with National Instruments for many years to recommend enhancements to LabVIEW that would enable Honeywell-Measurex to implement this very complex software system with graphical programming. Significant obstacles had to be



**Figure 1.14** MXProLine from Honeywell-Measurex is preconfigured with many user-interface screens such as this. It's probably the most elaborate LabVIEW application on the planet.

overcome, particularly in the areas of system configuration, initialization, and real-time network-transparent data distribution. These challenges were met by writing an external, real-time database in C++. It features very low overhead (microsecond access times), a hierarchical organization that follows the software organization as well as that of the physical plant, and the ability to share data with applications outside of LabVIEW. The package is data-driven to the point that the system can be fully configured without any LabVIEW programming. International installations can even switch languages on the fly! A typical MXProLine system can be configured to handle more than 100,000 variables (both physical I/O and computed values), with many operator workstations and I/O subsystems spread out over a plantwide network. More than 5000 VIs were written for this project. We know of no other LabVIEW-based system of this scale, and we're quite frankly blown away.

*This page intentionally left blank*

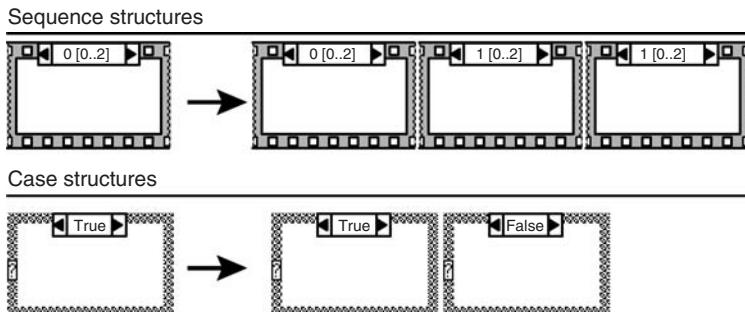
## Getting Started

Make no mistake about it: LabVIEW is a programming language, and a sophisticated one at that. It takes time to learn how to use it effectively. After you buy the program, you should go through the tutorial manual, and then hopefully you'll take the basic LabVIEW class, and later the advanced LabVIEW class; even though it costs money and takes time, it's *worth* it. Ask any of the MBAs in the front office; they know the value of training in terms of dollars and cents as well as worker productivity. Remember to keep working with LabVIEW; training goes stale without practice. There is a great deal of information to absorb. It's good to read, experiment, and then read some more. For the purposes of this book, we assume that you've been working with LabVIEW and know the basic concepts.

Another thing you should do is to spend plenty of time looking at the example VIs that come with the LabVIEW package. Collectively, the examples contain about 80 percent of the basic concepts that you really need to do an effective job, plus a lot of ready-to-use drivers and special functions. The rest you can get right here. Remember that this book is not a replacement for the user's manual. Yes, we know, you don't read user's manuals either, but it might be a good idea to crack the manuals the next time you get stuck. Most folks think that the manuals are pretty well written. And the online help fills in details for every possible function. What we cover in this book are important concepts that should make your programming more effective.

### About the Diagrams in This Book

It's hard to present LabVIEW through printed media. After all, it's an interactive graphical language, and the static nature of paper can obscure the workings of the underlying structure. In this book,



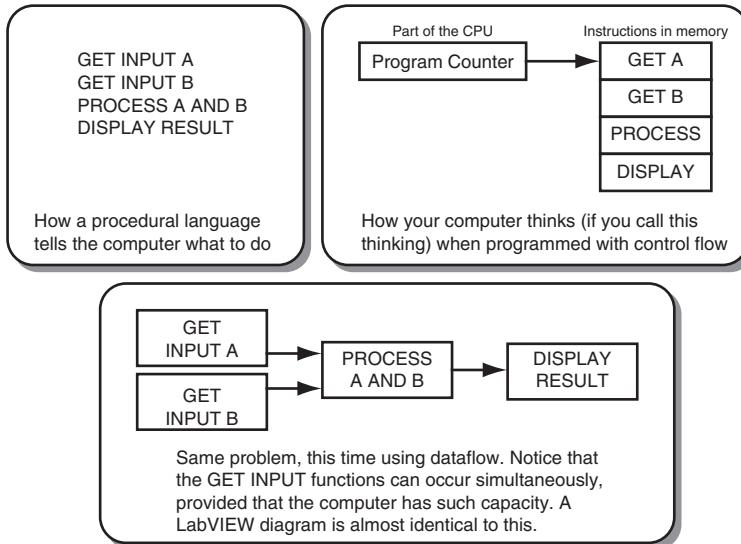
**Figure 2.1** A standard for printing Sequence and Case structures. This is but one compromise we must make when displaying LabVIEW programs on paper.

we've used some accepted conventions regarding images of LabVIEW diagrams and panels. One such convention is shown in Figure 2.1. Sequence structures and Case structures appear on the screen as single frames, but may in fact have several hidden frames, or subdiagrams. You view the subdiagrams by operating the control bar at the top. It's too bad you can't click on these pages. Instead, we'll just spread out the subdiagrams in a logical manner that should not cause too much confusion. For details on how you, too, can print and otherwise document LabVIEW programs, refer to the end of Chapter 9 under "Printing LabVIEW Panels and Diagrams."

## Sequencing and Data Flow

### CLAD

Regular textual programming languages (and most computers) are based on a concept called **control flow**, which is geared toward making things happen one step at a time with explicit ordering. The LabVIEW language, **G**, on the other hand, is known formally as a **dataflow** language. All it cares about is that each **node** (an object that takes inputs and processes them into outputs) have all its inputs available before executing. This is a new way of programming—one that you have to get used to before declaring yourself a qualified LabVIEW programmer. Although programming with dataflow is easy, it can be constraining because you have to operate within the confines of the dataflow model. This constraint also forces you to write programs that function well. Whenever you break from the dataflow model, you risk writing a program that misbehaves. The dataflow concept should be used to its best advantage at all times. This section discusses some of the concepts of dataflow programming with which you should be familiar.



**Figure 2.2** Compare and contrast control flow programming in a procedural language with dataflow programming in LabVIEW.

Figure 2.2 compares ordinary procedural programming (such as Basic or C) with dataflow programming. In all cases, the PROCESS A AND B step can execute only after GET A and GET B are completed. In the procedural language, we have forced GET A to happen before GET B. What if the source of data for B was actually ready to go before A? Then you would end up wasting time waiting for A. The dataflow version says nothing about whether GET A or GET B must go first. If the underlying code (courtesy of the LabVIEW compiler) is intelligent enough, the two GET tasks will overlap in time as required, enhancing throughput for this imaginary system.

LabVIEW permits you to have any number of different nodes on a diagram—all executing in parallel. Furthermore, the LabVIEW environment supports parallel execution, or multitasking, between multiple VIs, regardless of the capability of the operating system or computer. These capabilities give you great freedom to run various tasks asynchronously with one another without doing any special programming yourself. On the other hand, you sometimes need to force the sequence of execution to guarantee that operations occur in the proper order.

## LabVIEW under the Hood

Before we go too far, let's pop open the hood and take a look at how LabVIEW operates. This is an advanced topic and one not essential for

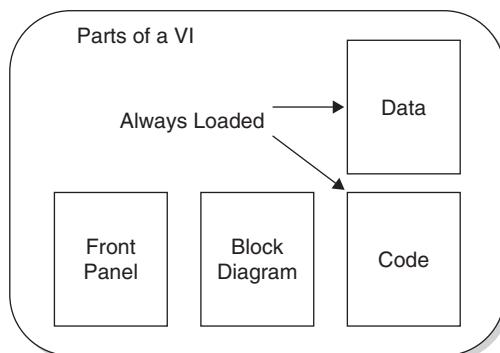
writing LabVIEW programs, but it helps you write better programs if you have an idea of what is going on. We're going to cover just the highlights. If you want greater in-depth coverage, there are several excellent presentations and applications notes on National Instruments' (NI's) Web site: AN 114, *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability*; AN 199, *LabVIEW and Hyper-Threading*; and Steve Roger's "Inside LabVIEW" presentation from NIWeek 2000. Incidentally, if you want to really understand LabVIEW, attend NIWeek. It is well worth your time.

### The parts of a VI

#### CLAD

We see VIs as front panels and block diagrams, but there is a lot that you don't see. Each VI is a self-contained piece of LabVIEW software with four key components (Figure 2.3):

- **Front panel.** The front panel code contains all the resources for everything on the front panel. This includes text objects, decorations, controls, and indicators. When a VI is used as a subVI, the front panel code is not loaded into memory unless the front panel is open or the VI contains a Property node.
- **Block diagram.** The block diagram is the dataflow diagram for the VI. The block diagram is not loaded into memory unless you open the block diagram or the VI needs to be recompiled.
- **Data.** The VI data space includes the default front panel control and indicator values, block diagram constants, and required memory buffers. The data space for a VI is always loaded into memory. Be careful



**Figure 2.3** VI code and data are always loaded into memory when the VI is loaded.

when you make current values default on front panel controls or use large arrays as block diagram constants; all that data is stored on disk in your VI.

- **Code.** This is the compiled machine code for your subVI. This is always loaded into memory. If you change platforms or versions of LabVIEW, the block diagram will be reloaded and the code will be recompiled.

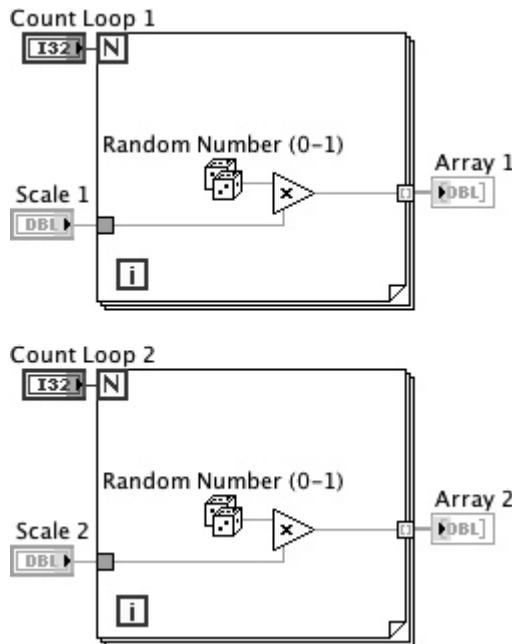
Each VI is stored as a single file with the four components above plus the linker information about the subVIs, controls, .dll, and external code resources that the VI needs. When a VI is loaded into memory, it also loads all its subVIs, and each subVI loads its subVIs, and so on, until the entire hierarchy is loaded.

### How VIs are compiled

There's an elegant term that the LabVIEW developers use for the code generated from your block diagram—it's called a *clump*. When you push the Run button on a new VI, LabVIEW translates the block diagram into clumps of machine code for your platform. The G compiler makes several passes through the block diagram to order the nodes into clumps of nodes. Execution order within each node is fixed at compile time by dataflow. As the compiler makes its passes, it determines which data buffers can be reused. Then it compiles the clumps into machine code. Here's an example based on Steve Roger's presentation: The block diagram in Figure 2.4 gets converted to three clumps of code. The first clump reads the controls and schedules the other two clumps for execution; then it goes to sleep. The other two clumps, one for each For Loop, run to completion, update their indicators, and reschedule the first clump for execution. The first clump then finishes the diagram, displays the data, and exits the VI. The traffic cop responsible for running the clumps of code is the LabVIEW execution engine.

### Multitasking, multithreaded LabVIEW

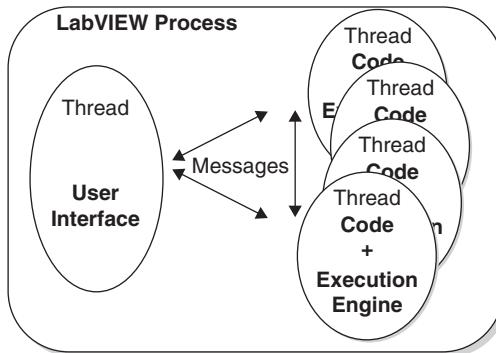
The LabVIEW execution engine runs each clump of code by using cooperative **multitasking**. The execution engine time slices each clump to make sure all the clumps get a chance to run. Each clump (in Figure 2.4, For Loop 1 or For Loop 2) either runs to completion or is switched out by the execution engine, and another clump runs. The execution engine proceeds on multitasking each clump of code until either your program ends or someone pushes the Abort button, which is always a nasty way to stop a program! Cooperative multitasking has been



**Figure 2.4** LabVIEW converts dataflow diagrams to “clumps” of machine code.

successfully used by LabVIEW since its birth, and it is still the process being used today. Beginning with LabVIEW 5, we’ve seen its effectiveness multiplied through **multithreading**. A **thread** is an independent program within a program that is managed by the operating system. Modern operating systems are designed to run threads efficiently, and LabVIEW receives a noticeable improvement from multithreading. Each LabVIEW thread contains a copy of the LabVIEW execution engine and the clumps of code that the engine is responsible for running. Within each thread the execution engine still multitasks clumps of code, but now the operating system schedules where and when each thread runs. On a multiprocessor system, the operating system can even run threads on different processors in parallel. Today LabVIEW on all major platforms is multithreaded and multitasking and able to take advantage of multiprocessor systems. Figure 2.5 illustrates the multiple threads within LabVIEW’s process. The threads communicate via a carefully controlled messaging system. Note that one thread is dedicated to the user interface. Putting the user interface into a separate thread has several advantages:

- Decoupling the user interface from the rest of your program allows faster loop rates within your code.



**Figure 2.5** Multithreading splits the LabVIEW process into multiple threads with one thread dedicated to the user interface. Each thread runs clumps of code with its own copy of the LabVIEW execution engine. Messages and other protected sharing mechanisms are used to communicate between threads.

- The user interface doesn't need to run any faster than the refresh rate on most monitors—less than 100 Hz.
- Decoupling the user interface from the rest of the program makes it possible to run the user interface on a separate computer. This is what happens in LabVIEW RT.

What does all this mean? It means the two For Loops in Figure 2.4 can run in parallel and you don't have to do anything other than draw them that way. The LabVIEW compiler generates code that runs in parallel, manages all the resources required by your program, spreads the execution around smoothly, and does it all transparently. Imagine if you had to build all this into a test system yourself. National Instruments has spent more than 20 years making sure LabVIEW is the best programming environment in test and measurement. You don't need to know the details of the execution engine or multithreading to program in LabVIEW, but we think it will make you a better programmer. Streamline your block diagram to take advantage of dataflow and parallelism, and avoid littering your block diagram with user-interface code.

## The LabVIEW Environment

We have already covered the difference between the front panel and the block diagram; but just to reiterate, the block diagram is where you put the nuts and bolts of your application, and the front panel contains your user interface. This is not going to be an exhaustive introduction

to LabVIEW—read the manuals and go through the tutorials for that. The rest of this chapter is an overview of LabVIEW and a jumping off point to other chapters in the book. Throughout this book we're going to focus on good programming practices and aspects that impact how LabVIEW runs your code.

### Front panel

The front panel is the user's window into your application. Take the time to do it right. Making a front panel easy to understand while satisfying complex requirements is one of the keys to successful virtual instrument development. LabVIEW, like other modern graphical presentation programs, has nearly unlimited flexibility in graphical user-interface design. But to quote the manager of a large graphic arts department, "We don't have a shortage of technology, we have a shortage of *talent!*" After years of waiting, we can finally refer you to a concise and appropriate design guide for user-interface design, Dave Ritter's book *LabVIEW GUI* (2001). Some industries, such as the nuclear power industry, have formal user-interface guidelines that are extremely rigid. If you are involved with a project in such an area, be sure to consult the required documents. Otherwise, you are very much on your own.

Take the time to look at high-quality computer applications and see how they manage objects on the screen. Decide which features you would prefer to emulate or avoid. Try working with someone else's LabVIEW application without getting any instructions. Is it easy to understand and use, or are you bewildered by dozens of illogical, unlabeled controls, 173 different colors, and lots of blinking lights? Observe your customer as he or she tries to use *your VI*. Don't butt in; just watch. You'll learn a lot from the experience, and fast! If the user gets stuck, you must fix the user interface and/or the programming problems. Form a picture in your own mind of good and bad user-interface design. Chapter 8, "Building an Application," has some more tips on designing a user interface.

One surefire way to create a bad graphical user interface (GUI) is to get carried away with colors, fonts, and pictures. When overused, they quickly become distracting to the operator. (Please don't emulate the glitzy screens you see in advertising; they are exactly what you *don't* want.) Instead, stick to some common themes. Pick a few text styles and assign them to certain purposes. Similarly, use a standard background color (such as gray or a really light pastel), a standard highlight color, and a couple of status colors such as bright green and red. Human factors specialists tell us that **consistency** and **simplicity** really are the keys to designing quality man-machine interfaces (MMIs).

Operators will be less likely to make mistakes if the layouts among various panels are similar. Avoid overlapping controls; multiple overlapping controls containing transparent pieces can really slow down the user interface.

Group logically related controls in the same area of the screen, perhaps with a surrounding box with a subtly different background color. You can see this technique on the panels of high-quality instruments from companies such as Tektronix and Hewlett-Packard. Speaking of which, those *real* instruments are excellent models for your *virtual* instruments.

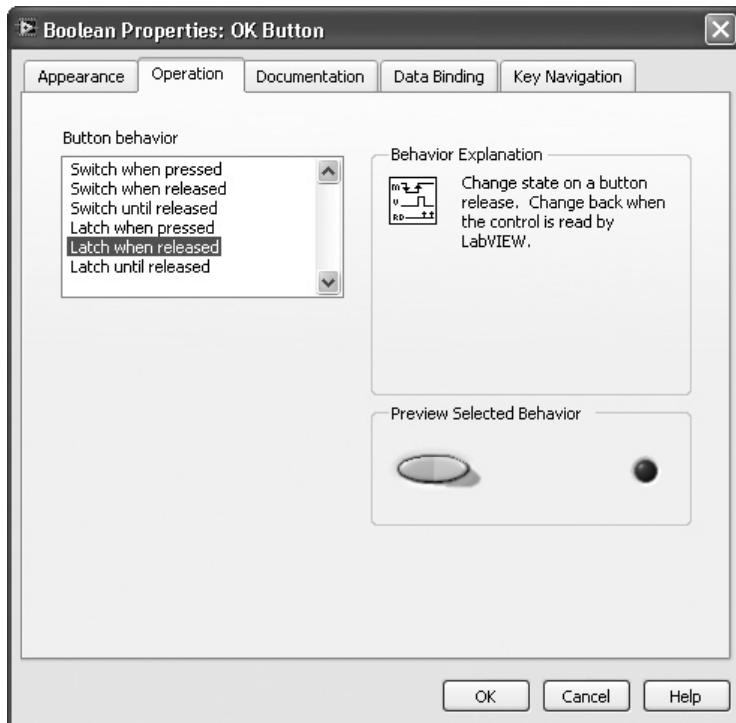
## Controls

LabVIEW's controls have a lot of built-in functionality that you should use for best effect in your application. And of course this built-in functionality runs in the user-interface thread. Quite a bit of display work, computational overhead, and user-interface functionality has already been done for you. All LabVIEW's graphs support multiple scales on both the X and Y axes, and all charts support multiple Y scales. Adjust the chart update modes if you want to see the data go from left to right, in sweep mode, update all at once in scope mode, or use the default strip chart mode going from right to left. Control properties are all easily updateable from the properties page. Figure 2.6 shows the property page for an OK button. Changing the button behavior (also called mechanical action) can simplify how you handle the button in your application. The default behavior for the OK button is "Latch when released," allowing users to press the buttons, change their minds, and move the mouse off the button before releasing. Your block diagram is never notified about the button press. On the other hand, latch when pressed notifies the block diagram immediately. Experiment with the mechanical behavior; there's a 1-in-6 chance the default behavior is right for your application.

## Property nodes



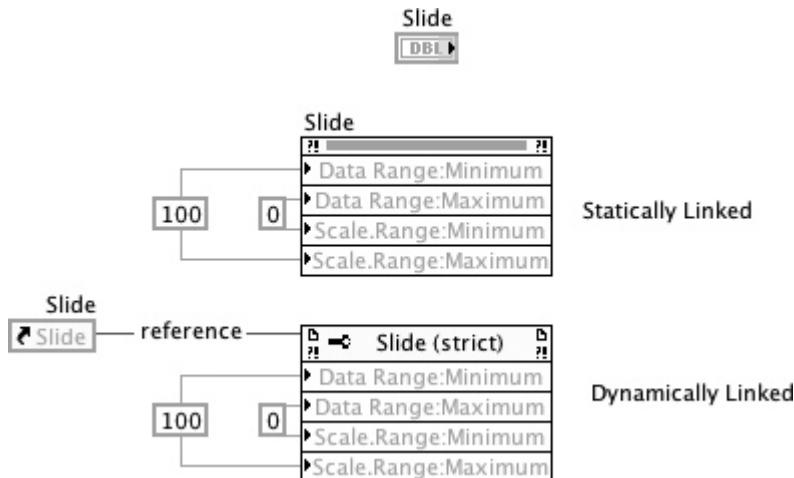
*Property nodes* are an extremely flexible way to manipulate the appearance and behavior of the user interface. You create a Property node by popping up on a control or indicator and choosing **Create . . . Property Node**. This creates a Property node that is **statically linked** to the control. You do not need to wire a reference to it for it to access the control. The disadvantage is that statically linked Property nodes can only reference front panel objects on their VI. Figure 2.7 shows both a statically linked and a **dynamically linked** Property node. With a **control reference** you can access properties of controls that aren't on



**Figure 2.6** Control properties are modified through the Property pages. Boolean buttons have six different mechanical actions that affect your block diagram.

the local front panel. This is a handy way to clean up a block diagram and generate some reusable user-interface code within a subVI. Every control and every indicator has a long list of properties that you can read or write. There can be multiple Property nodes for any given front panel item. A given Property node can be resized by dragging at a corner to permit access to multiple attributes at one time. Each item in the list can be either a read or a write, and execution order is sequential from top to bottom. We're not going to spend a great deal of time on Property nodes because their usage is highly dependent upon the particular control and its application. But you will see them throughout this book in various examples. Suffice it to say that you can change almost any visible characteristic of a panel item, and it's worth your time to explore the possibilities.

There are some performance issues to be aware of with Property nodes. Property nodes execute in the user-interface thread. Remember, in our discussion of dataflow and LabVIEW's execution system, how



**Figure 2.7** Statically linked Property nodes can only reference front panel objects within the scope of their VIs. Dynamically linked Property nodes use a control reference to access properties of controls that may be in other VIs.

each node depends on having all its data before it can proceed. Indiscriminately scattering Property nodes throughout your program will seriously degrade performance as portions of your block diagram wait while properties execute in the user interface. **A general rule of thumb is to never place Property nodes within the main processing loop(s).** Use a separate UI loop to interact with the front panel. Look at Chapter 3, “Controlling Program Flow,” to see how you can use Property nodes within the event structure.

Property nodes can increase the memory requirements of your application. Property nodes within a subVI will cause the subVI’s front panel to be loaded into memory. This may or may not be an issue, depending on what you have on your front panel, or how many subVIs have Property nodes on them. Try to limit the number of Property nodes and keep them contained with only a few VIs. You could even build a reusable suite of user-interface VIs by using Property nodes and references to front panel items.

One property that you want to avoid is the Value property. This is such a convenient property to use, but it is a performance hog. In LabVIEW 6 writing a control’s value with the Value property was **benchmarked at almost 200 times slower than using a local variable.** And a local variable is slower than using the actual control. LabVIEW is optimized for dataflow. Read and write to the actual controls and indicators whenever possible. You can do incredible things with Property nodes; just have the wisdom to show restraint.

### Block diagram

The block diagram is where you draw your program's dataflow diagram. For simple projects you might need only a few Express VIs. These are the "Learn LabVIEW in Three Minutes" VIs that salespeople are so fond of showing. They are great to experiment with, and you can solve many simple projects with just a few Express VIs. It is really easy to take measurements from a DAQ card and log it to disk compared to LabVIEW 2. However, when time comes to build a complete application to control your experiment, you'll probably find yourself scratching your head and wondering what to do. Read on. In Chapter 3, "Controlling Program Flow," we'll talk about design patterns that you can apply to common problems and build on.

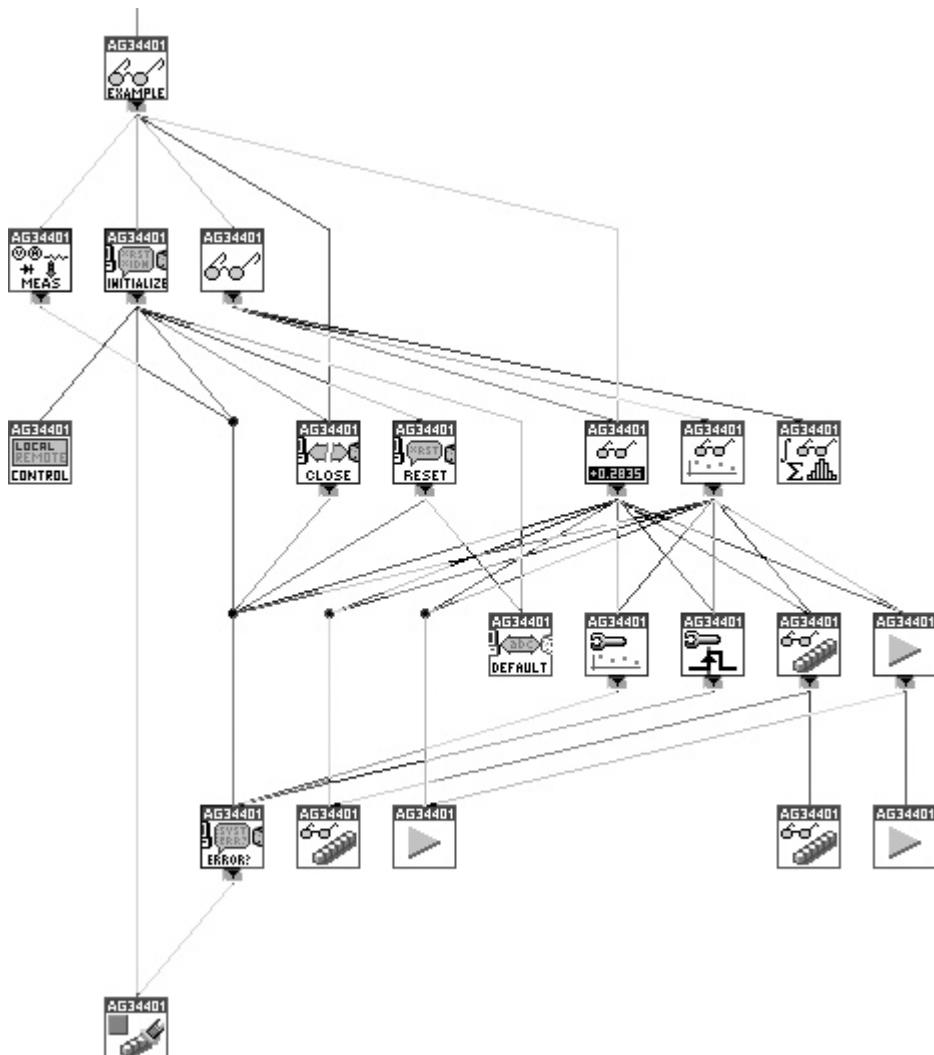
A large part of computer programming consists of breaking a complex problem into smaller and smaller pieces. This divide-and-conquer technique works in every imaginable situation from designing a computer program to planning a party. In LabVIEW the pieces are **subVIs**, and we have an advantage that other programming environments don't—each subVI is capable of being run and tested on its own. You don't have to wait until the application is complete and run against massive test suites. Each subVI can, and should, be debugged as it is developed. LabVIEW makes it easy to iteratively develop, debug, and make all the pieces work together.

### SubVIs

Approach each problem as a series of smaller problems. As you break it down into modular pieces, think of a one sentence statement that clearly summarizes the purpose. This one sentence is your subVI. For instance, one might be "This VI loads data from a series of transient recorders and places the data in an output array." Write this down with the labeling tool on the block diagram before you put down a single function; then build to that statement. If you can't write a simple statement like that, you may be creating a catchall subVI. Also write the statement in the VI description to help with documentation. Comments in the VI description will also show up in the context help window. Refer to Chapter 9, "Documentation," for more tips on documenting your VIs. Consider the reusability of the subVIs you create. Can the function be used in several locations in your program? If so, you definitely have a reusable module, saving disk space and memory. If the subVI requirements are *almost* identical in several locations, it's probably worth writing it in such a way that it becomes a universal solution—perhaps it just needs a mode control. Refer to Chapter 8, "Building an Application," for more help on general application design.

## Icons

Each subVI needs an icon. Don't rest content with the default LabVIEW icon; put some time and effort into capturing the essence of what the VI does. You can even designate terminals as being mandatory on the connector pane. They will show up in bold on the context help window. Also create an easily recognizable theme for driver VIs. Figure 2.8 shows the VI hierarchy for the Agilent 34401 Multimeter.



**Figure 2.8** VI hierarchy for Agilent 34401 Multimeter. The banner at the top of each icon gives the VIs a common theme. Each graphic succinctly captures VI function.

Refer to Chapter 11, “Instrument Driver Development Techniques,” for more information on building instrument drivers.

## Polymorphic VIs

### CLAD

Polymorphic VIs, introduced in LabVIEW 6, allow you to handle multiple data types with a single VI, thus going beyond the limitations of LabVIEW’s normal polymorphism. A polymorphic VI is like a container for a set of subVIs, each of which handles a particular data type. All those subVIs must have identical connector panes, but otherwise you’re free to build them any way you like, including totally different functionality. To create a new polymorphic VI, visit the File menu, choose New ..., and in the dialog box, pick Polymorphic VI. Up pops a dialog box in which you can select each of your subVIs. You can also give it an icon, but the connector pane is derived from the subVI. After you save it, your new polymorphic VI behaves just as any other VI, except that its inputs and outputs can adapt to the data types that you wire to them. This is called **compile-time polymorphism** because it’s accomplished before you run the VI. LabVIEW doesn’t really support polymorphism at run time (something that C++ does), although crafty users sometimes come up with strange type casting solutions that behave that way. Almost.

## Data

As you write your one-sentence description of each VI, consider the information that has to be passed from one part of the program to another. Make a list of the inputs and outputs. How do these inputs and outputs relate to other VIs in the hierarchy that have to access these items? Do the best you can to think of everything ahead of time. At least, try not to miss the obvious such as a channel number or error cluster that needs to be passed just about everywhere.

Think about the number of terminals available on the connector pane and how you would like the connections to be laid out. Is there enough room for all your items? If not, use **clusters** to group related items. Always leave a few uncommitted terminals on the connector pane in case you need to add an item later. That way, you don’t have to rewire everything.

## Clusters

### CLAD

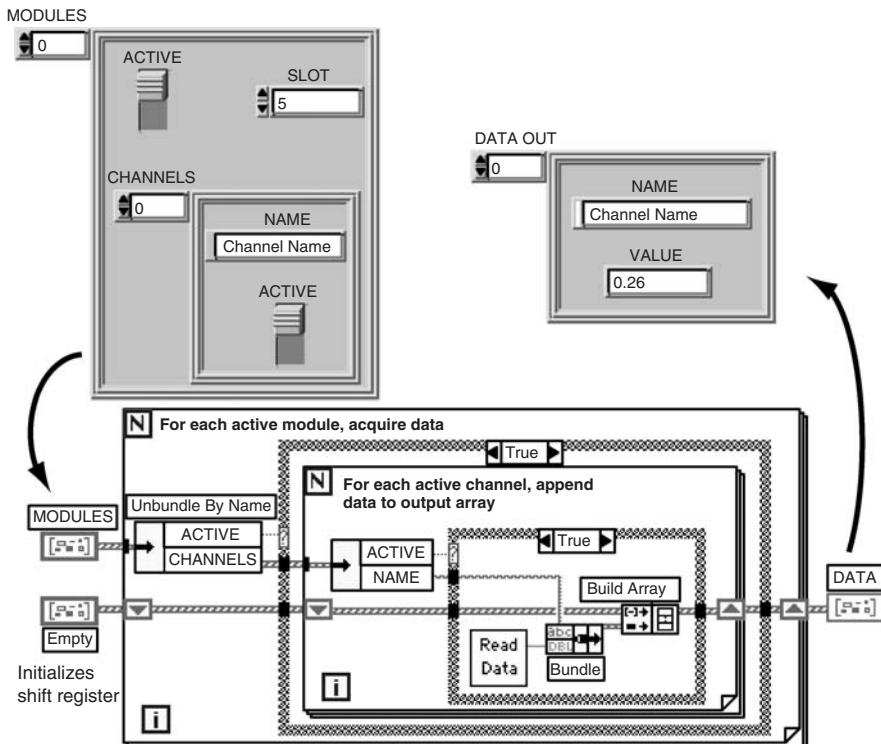
A **cluster** is conceptually the same as a *record* in Pascal or a *struct* in C. Clusters are normally used to group related data elements that are used in multiple places on a diagram. This reduces wiring

clutter—many items are carried along in a single wire. Clusters also reduce the number of terminals required on a subVI. When saved as a custom control with the **typedef** or **strict typedef** option (use the **Customize Control** feature, formerly known as the **Control Editor**, to create **typedefs**), clusters serve as data type definitions, which can simplify large LabVIEW applications. Saving your cluster as a **typedef** propagates any changes in the data structure to any code using the **typedef** cluster. Using clusters is good programming practice, but it does require a little insight as to when and where they are best employed. If you’re a novice programmer, look at the LabVIEW examples and the figures in this book to see how clusters are used in real life.

An important fact about a cluster is that it can contain only controls or indicators but not a mixture of both. This precludes the use of a cluster to group a set of controls and indicators on a panel. Use graphical elements from the Decorations palette to group controls and indicators. If you really need to read *and* write values in a cluster, local variables can certainly do the job. We would not recommend using local variables to continuously read and write a cluster because the chance for a race condition is very high. It’s much safer to use a local variable to initialize the cluster (just once), or perhaps to correct an errant input or reflect a change of mode. *Rule: For highly interactive panels, don’t use a cluster as an input and output element.*

Clusters make sense for other reasons. For instance, passing the name and calibration information for a signal along with its data in one cluster makes a nice unit that clarifies the program. Clustered data minimizes the number of wires on the diagram, too. **Data structures** such as clusters and arrays also make handling of data more efficient when they are closely coupled to the **algorithms** that you choose. An algorithm is a stepwise procedure that acts upon some associated data. Conversely, a well-chosen data structure reflects the organization of its associated algorithm. Getting them to work in harmony results in a clear LabVIEW diagram that runs at top speed. And as a bonus, the computer scientists will think you’re one of *them*.

A sample of this idea is shown in Figure 2.9. If there are many channels to process, keep them in an array. Having an array implies using a For Loop or While Loop to process that array. After the processing inside a loop, the logical output structure is another array. If the array needs to carry more information than simple numerics, make it a cluster array, as in the figure. The snippet of code shown here could readily stand on its own as a subVI, doing a task of data acquisition. Furthermore, it could easily be enhanced by adding items to the clusters, such as channel calibration information. If you think ahead to relate algorithms and data structures, that can certainly make your program



**Figure 2.9** The Modules data structure at the upper left implies the algorithm at the bottom because the way that the cluster arrays are nested is related to the way the For Loops are nested.

easier to understand and modify. The only drawback to complicated data structures may be performance. All that bundling, unbundling, and indexing of arrays does take some extra time and memory. An old rule for writing high-performance programs dictates that you do the smallest possible amount of data shuffling in high-speed loops and real-time VIs. For example, it may be worth splitting out the required data from a highly nested structure of clusters and arrays before it's time to enter a For Loop for processing.

### Typedefs



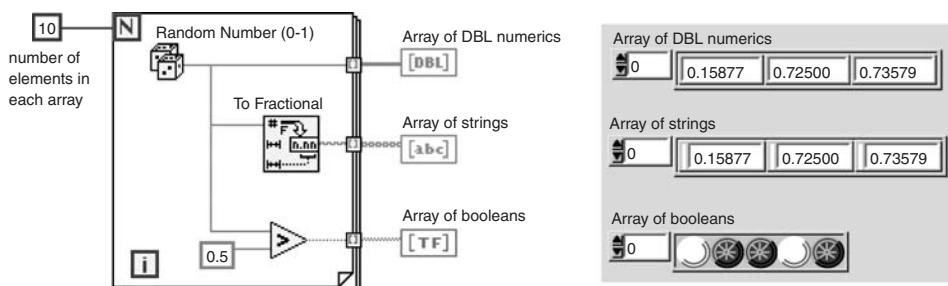
An important feature of LabVIEW that can help you manage complicated data structures is the **Type Definition**, which performs the same function as a **typedef** in C and a **Type statement** in Pascal. You edit a control by using the LabVIEW **Control Editor**, in which you

can make detailed changes to the built-in controls. Add all the items that you need in the data structure. Then you can save your new control with a custom name. An option in the Control Editor window is **Type Definition**. If you select this item, the control can't be reconfigured from the panel of any VI in which it appears; it can only be modified through the Control Editor. The beauty of this scheme is that the control itself now defines a data type, and changing it in one place (the Control Editor) automatically changes it in every location where it is used. If you don't use a Type Definition and you decide to change one item in the control, then you must manually edit every VI in which that control appears. Otherwise, broken wires would result throughout the hierarchy—a major source of suffering. As you can see, Type Definitions can save you much work and raise the quality of the final product.

## Arrays

### CLAD

Any time you have a series of numbers or any other data type that needs to be handled as a unit, the numbers probably belong in an **array**. Most arrays are one-dimensional (1D, a column or vector), a few are 2D (a matrix), and some specialized data sets require 3D or greater. LabVIEW permits you to create arrays of numerics, strings, clusters, and any other data type (except for arrays of arrays). The one requirement is that all the elements be of the same type. Arrays are often created by loops, as shown in Figure 2.10. For Loops are the best because they preallocate the required memory when they start. While Loops can't; LabVIEW has no way of knowing how many times a While Loop will cycle, so the memory manager will have to be called occasionally, slowing execution somewhat.



**Figure 2.10** Creating arrays by using a For Loop. This is an efficient way to build arrays with many elements. A While Loop would do the same thing, but without preallocating memory.

For Loops can also autoindex (execute sequentially for each element) through arrays. This is a powerful technique in test systems in which you want to increment through a list of test parameters. Pass the list as an array, and let your For Loop execute once for each value, or set of values. Refer to Chapter 4, “Data Types,” for more information on arrays and clusters.

## Debugging

Debugging is a part of life for all programmers (hopefully not *too* big a part). As you develop your application, squash each bug as it comes up. Never let them slip by, because it's too easy to forget about them later, or the situation that caused the bug may not come up again. Be sure to check each subVI with valid and invalid inputs. One thing you can always count on is that users will enter wrong values. LabVIEW has some tools and techniques that can speed up the debugging process. Read over the chapter of the LabVIEW manual on *executing and debugging VIs*. It has a whole list of debugging techniques. Here are a few of the important ones.

### See what the subVIs are up to

The first debugging technique is to open any subVIs that might be of interest in finding your problem. While a top-level VI executes, open and observe the lower-level VIs. Their panels will update each time they are called, allowing you to see any intermediate results. This is a good way to find errors such as swapped cluster elements or out-of-range values. If the value on one panel doesn't agree with the value on another panel, your wires are crossed somewhere. It's another reason for using Type Definitions on all your clusters.

There is one special trick to viewing the panels while running: If a subVI is set up to be **reentrant**, then you must double-click that subVI node on the diagram it is called from *while the caller is running*. Otherwise, you will just be looking at another copy of the reentrant code. Reentrant VIs have independent data storage areas allocated for each instance in which the VI is used.

You will also note that execution slows down when panels are open. This is caused by extra graphics updates. On a panel that is not displayed, the graphics don't have to be drawn, and that saves lots of execution time. There is also a saving in memory on VIs that are not displayed because LabVIEW must duplicate all data that is displayed in an indicator. If you are running a fast application where timing is critical, having extra panels open may murder your real-time response, so this debugging technique may not be feasible. On the other hand,

you can use the slowdown as a debugging tool in driver development. Sometimes, an instrument doesn't respond to commands as fast as you would think, and your VI can get out of sync. Having a lower-level VI's panel open can add just enough delay that the system starts working. Then you have a clue as to where to add some additional delay or logic.

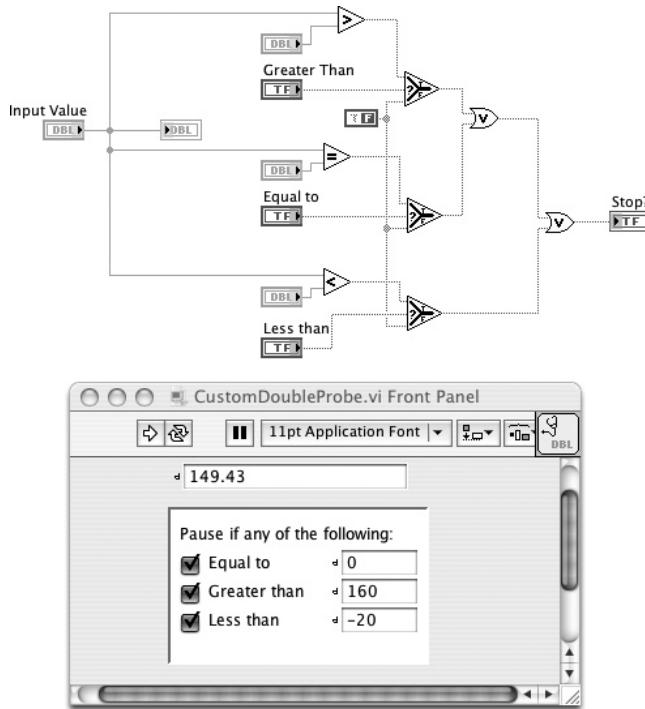
### Peeking at data

It is often helpful to look inside a VI at intermediate values that are not otherwise displayed. Even if no problems are evident, this can be valuable—one of those lessons that you can only learn the hard way, by having a VI that appears to be working, but only for the particular test cases that you have been using. During the prototyping stage, connect extra, temporary indicators at various points around the diagram as a sanity check. Look at such things as the iteration counter in a loop or the intermediate results of a complicated string search. If the values you see don't make sense, find out why. Otherwise, the problem will come back to haunt you.

While a VI is executing, you can click on any wire with the **probe** tool, or pop up on any wire and select *Probe* from the pop-up menu. A *windoid* (little window) appears that displays the value flowing through that wire. It works for all data types—even clusters and arrays—and you can even pop up on data in the probe window to prowl through arrays. You can have as many probe windows as you like, floating all over the screen, and the probes can be custom controls to display the data just the way you'd like. If the VI executes too fast, the values may become a blur; you may need to single-step the VI in that case (see the following section). Probes are great because you don't have to create, wire, and then later delete all kinds of temporary indicators. Use probes instead of execution trace when you want to view data in real time. A **Custom Probe** is a special subVI (Figure 2.11) you create that you can dynamically insert into a wire to see what is going on. Because a custom probe is a subVI you can put in any functionality you want. Right-click on the wire you want to probe, and select **Custom Probe >> New...** to create a new custom probe.

### One step at a time

Sometimes you just can't figure out what's happening when the VI runs at a full speed. In that case, try **single-stepping** the execution of your program. Single-stepping mode is entered by clicking on the **Pause button** [II]. When the VI runs, the diagram will open with the single-stepping controls displayed in the toolbar, and execution will be



**Figure 2.11** Custom probe and the probe’s block diagram. Custom probes are special VIs you can use to look at live data in a wire while the VI is executing. This custom probe is for a DBL data type and lets you conditionally pause the diagram. You can create custom probes with any functionality you want.

paused with some part of the diagram flashing. You then control execution with the following buttons:

**Step Into** allows the next node to execute. If the node is a subVI, the subVI opens in pause mode, and its diagram is displayed with part of its diagram flashing. This is a recursive type of debugging, allowing you to burrow down through the hierarchy.

**Step Over** lets you execute subVIs without opening them. For built-in functions, Step Over and Step Into are equivalent. This is my most used button for stepping through a big diagram.

**Step Out Of** lets you finish quickly. You can use this button to complete a loop (even if it has 10,000 iterations to go) or a Sequence structure without taking the VI out of pause mode. If you hold down the mouse button, you can control how many levels it steps out of.

If you have trouble remembering what each button does, turn on **Tip Strips** (use Tools >> Options Environment). Tip strips tell you in

a clear and context-sensitive manner what will happen with messages such as *Step into subVI “Get Data.”* During these single-stepping operations, you’re free to create probes anywhere in the hierarchy. For that matter, you can freely open subVIs, view panels, and do just about anything except edit a running VI.

### Execution highlighting

Another way to see exactly what is happening in a diagram is to enable **execution highlighting**. Click on the execution highlighting button  and notice that it changes to . Run the VI while viewing the diagram.

When execution starts, you can watch the flow of data. In the Debugging Preferences, you can enable data bubbles to animate the flow. Each time the output of a node generates a value, that value is displayed in a little box if you select the autoprobe feature in the Debugging Preferences. Execution highlighting is most useful in conjunction with single-stepping: You can carefully compare your expectations of how the VI works against what actually occurs. A favorite use for execution highlighting is to find an infinite loop or a loop that doesn’t terminate after the expected number of iterations.

While Loops are especially prone to execute forever. All you have to do is to accidentally invert the boolean condition that terminates the loop, and you’re stuck. If this happens in a subVI, the calling VI may be in a nonresponsive state. When you turn on execution highlighting or single-stepping, any subVI that is running will have a green arrow embedded in its icon: .

Sometimes a VI tries to run forever even though it contains no loop structures. One cause may be a subVI calling a Code Interface node (CIN) that contains an infinite loop. Reading from a serial port using the older Serial Port Read VI is another way to get stuck because that operation has no time-out feature. If no characters are present at the port, the VI will wait forever. (The solution to that problem is to call **Bytes at Serial Port** to see how many characters to read, then read that many. If nothing shows up after a period of time, you can quit without attempting to read. See Chapter 10, “Instrument Driver Basics,” for more information.) That is why you must always design low-level I/O routines and drivers with time-outs. In any case, execution highlighting will let you know which node is “hung.”

### Setting breakpoints

A **breakpoint** is a marker you set in a computer program to pause execution when the program reaches it. Breakpoints can be set on any subVI, node, or wire in the hierarchy by clicking on them with the

breakpoint tool, . A bold, red outline appears on the selected object. When the breakpoint is encountered during execution, the VI automatically enters pause mode and highlights the node that the program will execute next. You can click pause for one of the other single-stepping buttons to continue. To clear a breakpoint, point at it with the breakpoint tool and you'll notice that the tool changes to . Click, and the breakpoint is removed.

### Suspend when called

Another kind of breakpoint that works only with subVIs is **Suspend When Called**. You access this option through the VI Properties dialog or through the **SubVI Node Setup ...** dialog, accessed by popping up on the icon of a VI on a diagram. If a subVI's panel is not open when the breakpoint occurs, it will open automatically. The same thing happens when you have a numeric control or indicator set to suspend on a range error—a kind of conditional breakpoint. You might use this feature when something disastrous would happen if somebody entered an illegal value. Once the subVI is suspended, it will not proceed until you click one of the following buttons on the control bar:

 is the **Run** button, which runs the subVI as if it were a top-level VI being run on its own. You can run it over and over, changing the input values and observing the results. A really cool thing you can do while in this state (and no other) is to edit the values of *indicators*. If the VI runs for a long time and you need to abort it, you can click the Abort button , and it will stop the entire hierarchy.

 is the **Return to caller** button. Click this after you have finished fiddling with any values on the panel. Control returns to the calling VI, whether or not you have clicked the Run button (remember to do so if you wish to have the subVI compute new values). The combination of these two buttons permits you to completely override the action of a subVI, which is handy when you want to perform a test with artificial data without modifying any diagrams.

## Calling Other Code

### CINs

Sometimes, programming in C is simply the best way to solve a problem. In that case, you can use **Code Interface nodes (CINs)** which are written in C and link nicely into LabVIEW. They are ideal when you want to import an existing code because you can do so almost directly. CINs are also needed for direct access to your computer's system routines, graphics toolbox, and the like. They can be used to speed up

areas where LabVIEW may not be as efficient, that is, conversion of large arrays of numbers to strings with intricate formatting. In that case you are forced to chain several string functions together inside a loop with several string concatenations for each iteration. In that case, LabVIEW has to call the memory manager on every iteration of the loop (maybe several times on each iteration), and the resultant process is very slow. In C, you can compute the final string size, allocate memory just once, and write to the output string very efficiently. The problem with CINs is that the average user can't maintain them because they require a compiler for, and knowledge of, another language. Furthermore, CINs must be recompiled before they can run on a different computer. That's why they're called *sins*. They do make a great escape hatch for situations in which you have no other choice.

### Dynamic link libraries

Another way to access external code modules is through the **Call Library** node. It looks like a CIN on the diagram, but it's configured through a pop-up dialog through which you define the data type and other attributes of each input and output. One of the items you have to configure is whether LabVIEW can use the library reentrantly. LabVIEW needs to know if multiple callers can use the piece of code simultaneously or if access needs to be synchronous. If the Dynamic link library (DLL) function is not reentrant, then LabVIEW will run it in the user-interface thread. You can tell from color of the Call library function on the block diagram how the call is configured. Non-reentrant calls are orange and reentrant calls are yellow. The inputs and outputs of the Call Library node have a one-to-one correspondence with the inputs and outputs of the external module that you are calling. One "gotcha" of DLL programming is that LabVIEW only runs initialization code in a DLL one time when LabVIEW first loads the DLL. If you're noticing strange behavior in your DLL calls, this is something to check.

Windows makes extensive use of **Dynamic Link Libraries (DLLs)** to encapsulate drivers and various system functions. Other platforms use **shared libraries** for adding greater system functionality. Almost any compiler can generate a DLL or shared library—and you're not limited to C. As long as you know what the parameters are, Call Library will make the connection.

### Programming by Plagiarizing

A secret weapon for LabVIEW programming is the use of available example VIs as starting points. Many of your problems have already been solved by someone else, but the trick is to get hold of that code.

The examples and driver libraries that come with LabVIEW are a gold mine, especially for the beginner. Start prowling these directories, looking for tidbits that may be useful. Install LabVIEW on your home computer. Open up every example VI and figure out how it works. If you have access to a local LabVIEW user group, be sure to make some contacts and discuss your more difficult problems. The online world is a similarly valuable resource. We simply could not do our jobs any more without these resources.

The simplest VIs don't require any fancy structures or programming tricks. Instead, you just plop down some built-in functions, wire them up, and that's it. This book contains many programming constructs that you will see over and over. The examples, utilities, and drivers can often be linked to form a usable application in no time at all. Data acquisition using plug-in multifunction boards is one area where you rarely have to write your own code. Using the examples and utilities pays off for file I/O as well. We mention this because everyone writes her or his own file handlers for the first application. It's rarely necessary, because 90 percent of your needs are probably met by the file utilities that are already on your hard disk: **Write to Text File** and **Read From Text File**. The other 10 percent you can get by modifying an example or utility to meet your requirements. Little effort on your part and lots of bang for the buck for the customer. That's the way it's supposed to work.

## Bibliography

- AN 114, *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability*, www.ni.com, National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2000.
- AN 199, *LabVIEW and Hyper-Threading*, www.ni.com, National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2004.
- Brunzie, Ted J.: "Aging Gracefully: Writing Software that Takes Changes in Stride," *LabVIEW Technical Resource*, vol. 3, no. 4, Fall 1995.
- Fowler, Gregg: "Interactive Architectures Revisited," *LabVIEW Technical Resource*, vol. 4, no. 2, Spring 1996.
- Gruggett, Lynda: "Getting Your Priorities Straight," *LabVIEW Technical Resource*, vol. 1, no. 2, Summer 1993. (Back issues are available from LTR Publishing.)
- "Inside LabVIEW," NIWeek 2000, Advanced Track 1C. National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2000.
- Johnson, Gary W. (Ed.): *LabVIEW Power Programming*, McGraw-Hill, New York, 1998.
- Ritter, David: *LabVIEW GUI*, McGraw-Hill, New York, 2001.

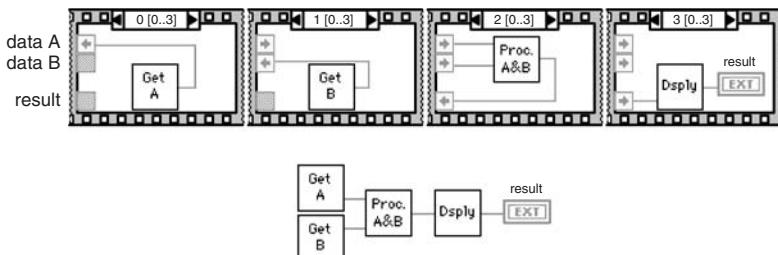
## Controlling Program Flow

Now we are ready to look at the nuts and bolts of LabVIEW programming, beginning with the fundamentals of how to control LabVIEW flow. A fundamental theme of LabVIEW programming is dataflow and executing nodes based on data dependency. A well-written LabVIEW program flows in a left-to-right and top-to-bottom order, but the execution order is always governed by dataflow. Without dataflow it is impossible to determine execution order. In fact, if you place independent items on the block diagram without any data dependency between them, they will multitask in parallel. Multitasking is a powerful concept made simple in LabVIEW. However, usually you need to order a program's execution—random execution in a test and measurement system is not a good idea. Dataflow can be forced through sequence frames, or through a common thread flowing through all the block diagram items.

### Sequences

**CLAD**

The simplest way to force the order of execution is to use a Sequence structure as in the upper example of Figure 3.1. Data from one frame is passed to succeeding frames through **Sequence local variables**, which you create through a pop-up menu on the border of the Sequence structure. In a sense, this method avoids the use of (and advantages of) dataflow programming. You should try to avoid the overuse of Sequence structures. LabVIEW has a great deal of inherent parallelism, where GET A and GET B could be processed simultaneously. Using a sequence guarantees the order of execution but prohibits parallel operations. For instance, asynchronous tasks that use I/O devices (such as GPIB and



**Figure 3.1** In the upper example, Sequence structures force LabVIEW to *not* use dataflow. Do this only when you have a good reason. The lower example shows the preferred method.

serial communications and plug-in boards) can run concurrently with CPU-bound number-crunching tasks. Your program may actually execute faster if you can add parallelism by reducing the use of Sequence structures. Note that Sequence structures add *no code or execution overhead* of their own. Perhaps the worst features of Sequence structures are that they tend to hide parts of the program and that they interrupt the natural left-to-right visual flow. The lower example in Figure 3.1 is much easier to understand and permits parallel execution of GET A and GET B.

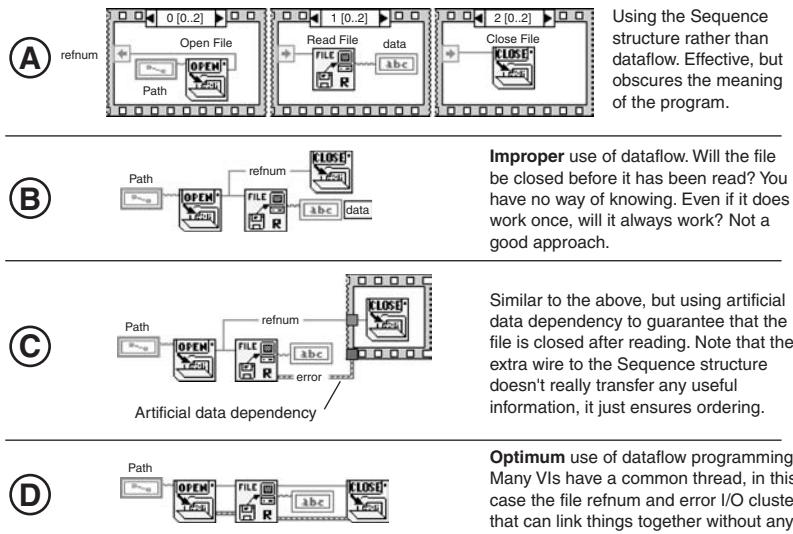
Sequencing is mandatory in some problems and can clarify the program structure by grouping logically connected operations into neat frames. Don't get carried away with the *avoidance* of Sequence structures. An alternative to the stacked Sequence structure is the flat Sequence structure introduced in LabVIEW 8. The flat Sequence structure executes frames in a nice left-to-right order and does not obscure dataflow. Good use of dataflow results in a clear, single-page main program that is easy to understand.

## Data Dependency

### CLAD

A fundamental concept of dataflow programming is **data dependency**, which says that a given node can't execute until *all* its inputs are available. So in fact, you can write a program using only dataflow, or not. Let's look at an example that is very easy to do with a sequence, but is better done without. Consider Figure 3.2, which shows four possible solutions to the problem where you need to open a file, read from it, and then close it.

Solution A uses a stacked sequence. Almost everybody does it this way the first time. Its main disadvantage is that you have to flip through the subdiagrams to see what's going on. Now, let's do it with dataflow,



**Figure 3.2** Four ways to open, read, and close a file, using either Sequence or dataflow programming methods. Example D is preferred.

where all the functions are linked by wires in some logical order. A problem arises in solution B that may not be obvious, especially in a more complicated program: Will the file be read before it is closed? Which function executes first? Don't assume top-to-bottom or left-to-right execution when no data dependency exists! Make sure that the sequence of events is explicitly defined when necessary. A solution is to create **artificial data dependency** between the **Read File** and the **Close File** functions, as shown in example C. We just connected one of the outputs of the Read File function (any output will do) to the border of the Sequence structure enclosing the Close File. That single-frame sequence could also have been a Case structure or a While Loop—no matter: It's just a container for the next event. The advantage of this style of programming is clarity: The entire program is visible at first glance. Once you get good at it, you can write many of your programs in this way.

## Adding Common Threads

Going further with the idea of data dependency, you can build flow control right into your subVIs as demonstrated in example D. If you make a collection of subVIs that are frequently used together, give them all a common input/output terminal pair so that they can be chained together

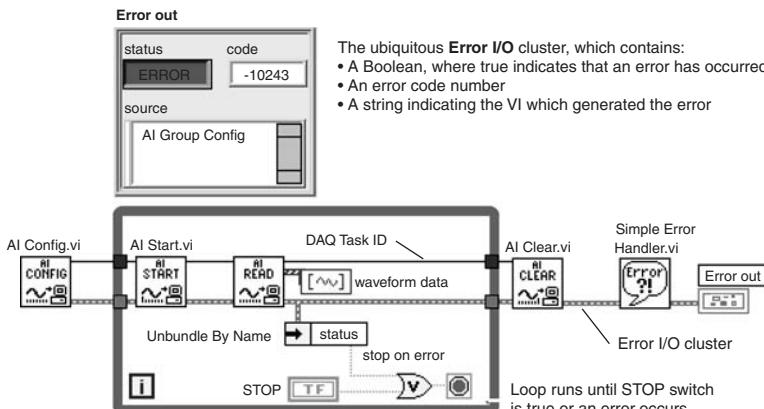
without requiring Sequence structures. In the case of LabVIEW's file I/O functions, it turns out that the file refnum is duplicated by the **Read File** function, and it can be passed along to the **Close File** function. Thus, the read operation has to be completed before it permits the file to be closed. The problem is solved very neatly.

### CLAD

A good common thread is an error code, since just about every operation that you devise probably has some kind of error checking built into it, particularly those that do I/O operations. Each VI should test the incoming error and not execute its function if there is an existing error. It should then pass that error (or its own error) to the output. You can assemble this error information into a cluster containing a numeric error code, a string containing the name of the function that generated the error, and an error boolean for quick testing. This technique, which is a universal standard first promoted by Monnie Anderson at National Instruments, is called **Error I/O**. Particular examples are the data acquisition (DAQ) library functions (Figure 3.3) GPIB, VISA, serial, and file I/O. Error I/O is discussed in detail in Chapter 10, "Instrument Driver Basics."

## Looping

Most of your VIs will contain one or more of the two available loop structures, the **For Loop** and the **While Loop**. Besides the obvious use—doing an operation many times—there are many nonobvious



**Figure 3.3** A cluster containing error information is passed through all the important subVIs in this program that use the DAQ library. The While Loop stops if an error is detected, and the user ultimately sees the source of the error displayed by the Simple Error Handler VI. Note the clean appearance of this style of programming.

ways to use a loop (particularly the While Loop) that are helpful to know about. We'll start with some details about looping that are often overlooked.

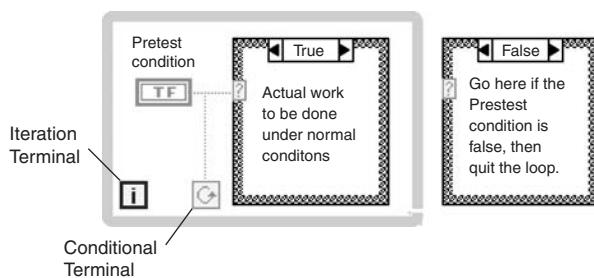
## While LOOPS

### CLAD

The While Loop is one of the most versatile structures in LabVIEW. With it, you can iterate an unlimited number of times and then suddenly quit when the boolean **conditional terminal** becomes True. You can also pop up on the conditional terminal and select Stop If False. If the condition for stopping is always satisfied (for instance, Stop If False wired to a False boolean constant), the loop executes exactly *one time*. If you put uninitialized shift registers on one of these one-trip loops and then construct your entire subVI inside it, the shift registers become a memory element between calls to the VI. We call this type of VI a **functional global**. You can retain all sorts of status information in this way, such as knowing how long it's been since this subVI was last called (see the section on shift registers that follows).

**Pretest While Loop.** What if you need a While Loop that does not execute *at all* if some condition is False? We call this a *pretest* While Loop, and it's really easy to do (Figure 3.4). Just use a Case structure to contain the code that you might not want to execute.

**Graceful stops.** It's considered bad form to write an infinite loop in LabVIEW (or any other language, for that matter). Infinite loops run forever, generally because the programmer told them to stop only when  $2 + 2 = 5$ , or something like that. Gary remembers running Fortran programs on the old Cyber 175 back at school, where an infinite loop was rather hard to detect from a timesharing terminal—there was always



**Figure 3.4** A pretest While Loop. If the pretest condition is False, the False frame of the Case structure is executed, skipping the actual work to be done that resides in the True case.

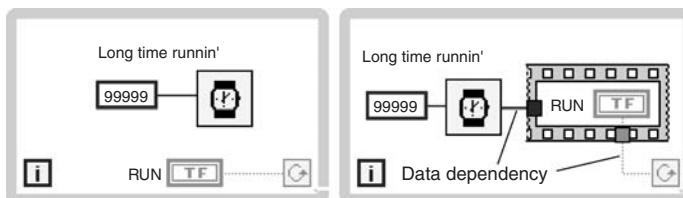
an unpredictable delay before the friendly prompt came back, regardless of how long your program took to execute. Trouble was, some folks had to pay real money for computer time, and that computer cost about \$1000 per hour to run! The hazard in LabVIEW is that the only way to stop an infinite loop is to click the Abort icon  in the toolbar. I/O hardware could be left in an indeterminate state. As a minimum, you need to put a boolean switch on the front panel, name it STOP, and wire it to the conditional terminal. The boolean control palette even has buttons prelabeled STOP. Use whatever is appropriate for your experiment.

A funny thing happens when you have a whole bunch of stuff going on in a While Loop: It sometimes takes a long time to stop when you press that STOP boolean you so thoughtfully included. The problem (and its solution) is shown in Figure 3.5. What happens in the left frame is that there is no data dependency between the guts of the loop and the RUN switch, so the switch may be read before the rest of the diagram executes. If that happens, the loop will go around one more time before actually stopping. If one cycle of the loop takes 10 minutes, the user is going to wonder why the RUN switch doesn't seem to do anything. Forcing the RUN boolean to be evaluated as the last thing in the While Loop cures this extra-cycle problem.

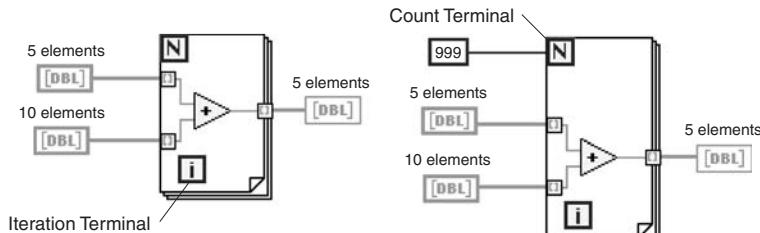
## For Loops

### CLAD

Use the **For Loop** when you definitely know how many times a subdiagram needs to be executed. Examples are the building and processing of arrays and the repetition of an operation a fixed number of times. Most of the time, you process arrays with a For Loop because LabVIEW already knows how many elements there are, and the **auto-indexing** feature takes care of the iteration count for you automatically: All you have to do is to wire the array to the loop, and the number of iterations (count) will be equal to the number of elements in the array.



**Figure 3.5** The left example may run an extra cycle after the RUN switch is turned off. Forcing the switch to be evaluated as the last item (right) guarantees that the loop will exit promptly.



**Figure 3.6** The rule of For Loop limits: The smallest count always wins, whether it's the N terminal or one of several arrays.

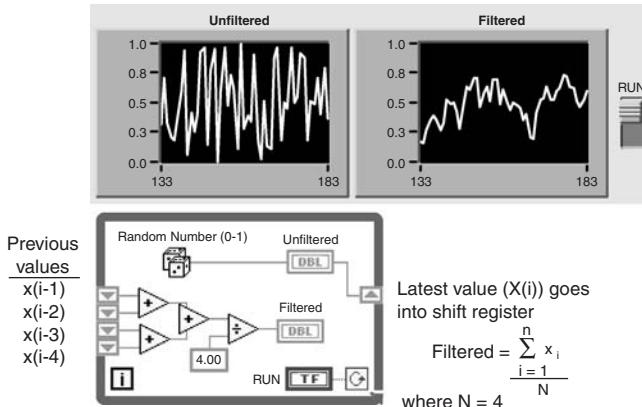
But what happens when you hook up more than one array to the For Loop, each with a different number of elements? What if the count terminal **N** is also wired, but to yet a different number? Figure 3.6 should help clear up some of these questions. *Rule: The smaller count always wins.* If an empty array is hooked up to a For Loop, that loop will never execute.

Also, note that there is no way to abort a For Loop. In most programming languages, there is a GOTO or an EXIT command that can force the program to jump out of the loop. Such a mechanism was never included in LabVIEW because it destroys the dataflow continuity. Suppose you just bailed out of a For Loop from inside a nested Case structure. To what values should the outputs of the Case structure, let alone the loop, be set? Every situation would have a different answer; it is wiser to simply enforce good programming habits. If you need to escape from a For Loop, use a While Loop instead! By the way, try popping up (see your tutorial manual for instructions on popping up on your platform) on the border of a For Loop, and use the Replace operation; you can instantly change it to a While Loop, with no rewiring. The reverse operation also works. You can even swap between Sequence and Case structures, and LabVIEW preserves the frame numbers. Loops and other structures can be removed entirely, leaving their contents wired in place so far as possible, by the same process (for example, try the Remove For Loop operation). See the LabVIEW user's manual for these and other cool editing tricks that can save you time.

## Shift registers

### CLAD

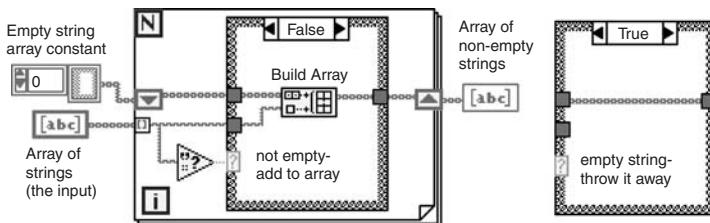
**Shift registers** are special local variables or memory elements available in For Loops and While Loops that transfer values from the completion of one iteration to the beginning of the next. You create them by popping up on the border of a loop. When an iteration of the loop completes, a value is written to the shift register—call this the *n*th value.



**Figure 3.7** Using a shift register to perform a simple moving average on random numbers. Note the difference in the graphs of filtered and unfiltered data. This is a very simple case of a finite impulse response filter.

On the next iteration, that value is available as a source of data and is now known as the  $(n - 1)$ st value. Also, you can add as many terminals as you like to the left side, thus returning not only the  $(n - 1)$ st value, but also  $n - 2$ ,  $n - 3$ , and so on. This gives you the ability to do digital filtering, modeling of discrete systems, and other algorithms that require a short history of the values of some variable. Any kind of data can be stored in a shift register—they are **polymorphic**. Figure 3.7 is an implementation of a simple *finite impulse response* (FIR) filter, in this case a moving averager that computes the average of the last four values. You can see from the strip charts how the random numbers have been smoothed over time in the filtered case.

Besides purely mathematical applications like this one, there are many other things you can do with shift registers. Figure 3.8 is a very



**Figure 3.8** Weeding out empty strings with a shift register. A similar program might be used to select special numeric values. Notice that we had to use Build Array inside a loop, a relatively slow construct, but also unavoidable.

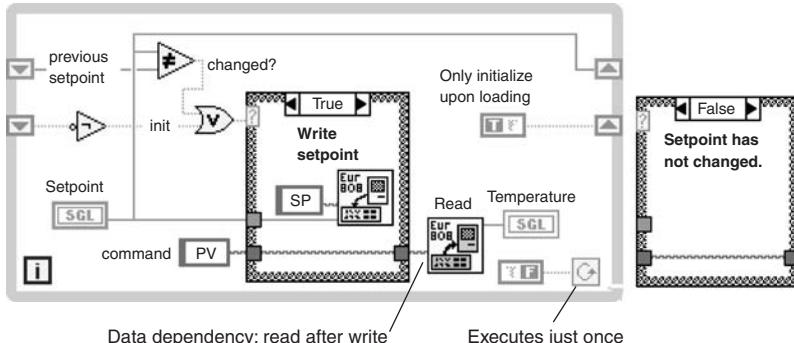
common construct where an array is assembled based on some conditional testing of the incoming data. In this case, it's weeding out empty strings from an incoming array. This program is frequently seen in configuration management programs. In that case, a user is filling out a big cluster array that contains information about the various channels in the system. Leaving a channel name empty implies that the channel is unused and should be deleted from the output array. The Build Array function is located inside a Case structure that checks to see if the current string array element is empty. To initialize the shift register, create a one-dimensional (1D) string array constant on the diagram by popping up on the shift register terminal and selecting Create Constant.

### Uninitialized shift registers

In Figure 3.8, what would happen if that empty string array constant already had something in it? That data would appear in the output array *ahead* of the desired data. That's why we want an *empty* array for initialization. What happens if the empty string array is left out altogether? The answer is that the first time the program runs after being loaded, the shift register is in fact empty and works as expected. But the shift register then *retains its previous contents until the next execution*. Every time you ran this modified VI, the output string would get bigger and bigger (a good way to make LabVIEW run out of memory, by the way). All shift registers are initialized at compile time as well: Arrays and strings are empty, numerics are zero, and booleans are False. There are some important uses for uninitialized shift registers that you need to know about.

#### CLAD

First, you can use uninitialized shift registers to keep track of state information between calls to a subVI. This is an extremely powerful technique and should be studied closely. You will see it dozens of times in this book. In Figure 3.9, a shift register saves the previous value of the set point front panel control and compares it to the current setting. If a change of value has occurred, the new setting is sent to a temperature controller via the serial port. By transmitting only when the value has changed, you can reduce the communications traffic. This technique, known as **change-of-state detection**, can make a VI act more intelligently toward operator inputs. Use a second (boolean) shift register to force the transmission of the set point the first time this VI is run. Since the boolean shift register is False when loaded, you can test for that condition as shown here. Then write True to the shift register to keep the VI from performing the initialization operation again.

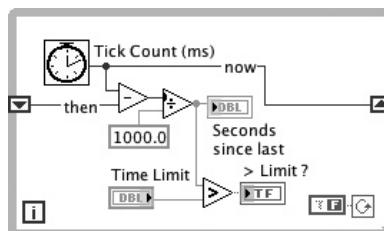


**Figure 3.9** This VI writes a new set point to a Eurotherm 808 temperature controller only when the user has changed the value. It then reads the current temperature. The upper shift register saves the previous set point value for change-of-state detection. The lower shift register is used for initialization to guarantee that a set point will always be written when the VI is first run.

Note that the conditional terminal is wired with a False constant, which means that the contents of the loop execute only once each time the VI is called. Such a construct is of little use as a top-level VI. Instead, its intended use is as a more intelligent subVI—one with **state memory**. That is, each time this subVI is called, the contents of the shift register provide it with logical information regarding conditions at the end of the previous iteration.

This allows you to program the subVI to take action based not only on the current inputs, but on previous inputs as well. This concept will be extrapolated into powerful **state machines** in subsequent chapters.

The shift register in Figure 3.10 keeps track of the elapsed time since the subVI was last called. If you compare the elapsed time with



**Figure 3.10** This example shows how to use an uninitialized shift register to determine how long it's been since this subVI was last called. This makes the subVI time aware and useful for data logging, watchdog timing, or time-dependent calculations.

a desired value, you could use this to trigger a periodic function such as data logging or watchdog timing. This technique is also used in the PID (proportional integral derivative) Toolkit control blocks whose algorithms are time-dependent.

## Globals

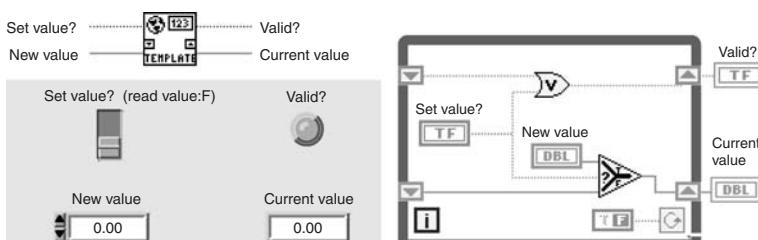
### Global and local variables

#### CLAD

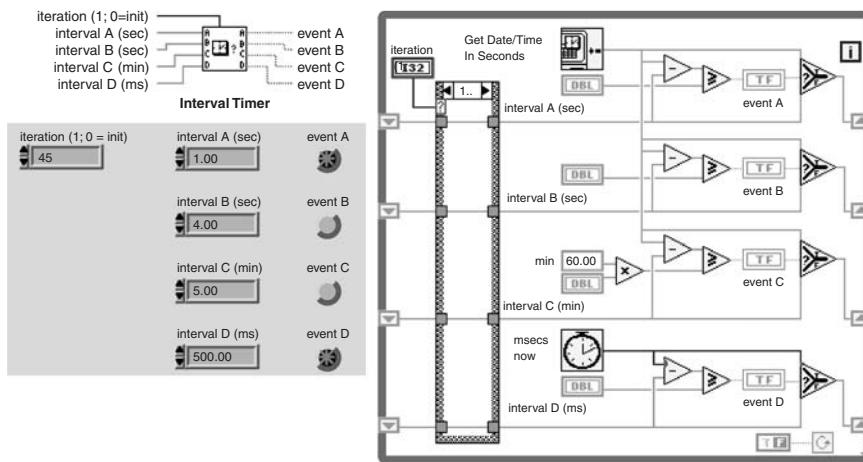
Another, very important use for uninitialized shift registers occurs when you create **global variable VIs**, also called **Functional Globals**. In LabVIEW, a variable is a wire connecting two objects on a diagram. Since it exists only on one diagram, it is by definition a local variable. By using an uninitialized shift register in a subVI, two or more calling VIs can share information by reading from and writing to the shift register in the subVI. This is a kind of global variable, and it's a very powerful notion. Figure 3.11 shows what the basic model looks like.

If **Set Value** is True (write mode), the input value is loaded into the shift register and copied to the output. If **Set Value** is False (read mode), the old value is read out and recycled in the shift register. The **Valid** indicator tells you that something has been written; it would be a bad idea to read from this global and get nothing when you expect a number. Important note: For each global variable VI, you must create a distinct VI with a *unique name*.

There are several really nice features of global variables. First, they can be read or written any time, any place, and all callers will access the same copy of the data. There is no data duplication. This includes multiple copies on the same diagram and, of course, communications between top-level VIs. This permits asynchronous tasks to share information. For instance, you could have one top-level VI that just scans



**Figure 3.11** A global variable VI that stores a single numeric value. You could easily change the input and output data type to an array, a cluster, or anything else.



**Figure 3.12** A time interval generator based on uninitialized shift registers. Call this subVI with one of the Event booleans connected to a Case structure that contains a function you wish to perform periodically. For instance, you could place a strip chart or file storage function in the True frame of the Case structure.

your input hardware to collect readings at some rate. It writes the data to a global array. Independently, you could have another top-level VI that displays those readings from the global at another rate, and yet another VI that stores the values. This is like having a global database or a client-server relationship among VIs.

Because these global variables are VIs, they are more than just memory locations. They can perform processing on the data, such as filtering or selection, and they can keep track of time as the **Interval Timer VI does** in Figure 3.12. Each channel of this interval timer compares the present time with the time stored in the shift register the last time when the channel's **Event** output was True. When the **Iteration** input is equal to zero, all the timers are synchronized and are forced to trigger. This technique of initialization is another common trick. It's convenient because you can wire the **Iteration** input to the iteration terminal, in the calling VI, which then initializes the subVI on the first iteration. We use this timer in many applications throughout this book.

As you can see, functional globals can store multiple elements of different types. Just add more shift registers and input/output terminals. By using clusters or arrays, the storage capacity is virtually unlimited.

LabVIEW has built-in global variables as well (discussed in the next section). They are faster and more efficient than functional global variables and should be used preferentially. However, you can't embed any intelligence in them since they have no diagram; they are merely data storage devices.

There is one more issue with functional global variables: performance. Scalar types including numerics and booleans are very fast and limited only by LabVIEW's subVI calling overhead. Arrays or clusters of numerics and booleans are a little slower, but the number of elements is what is most important. The real killer comes when your global variable carries clusters or arrays that contain strings. This requires the services of the memory manager every time it is called, and thus the performance is relatively poor (almost 10 times slower than a comparably sized collection of numerics).

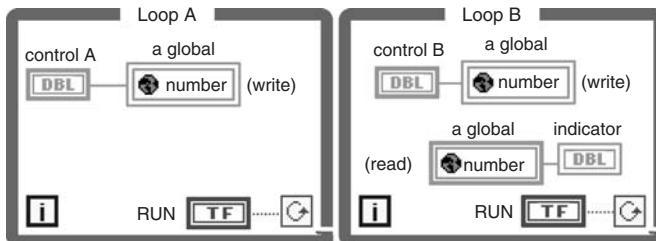
### Built-in global variables—and their hazards

Starting with LabVIEW 3, global variables are built in. To create a global variable, you select it from the Structures subpalette of the Function palette, place it on the diagram, and then double-click on it to open and edit its front panel, exactly as with any subVI. You can place any number of controls on the panel, name them, and then save the global variable with a name—again, just as with any subVI. Back on your main diagram, pop up on the global variable, select one of the controls that you put in the global, and then choose whether you wish to read or write data. Finally, wire it to the appropriate source or destination. At any time, you can open the panel of the global variable to view or change its contents. That can be very helpful during a debugging session.

The difference between built-in globals and the functional globals you make yourself with a subVI is that the built-in ones are not true VIs and as such cannot be programmed to do anything besides simple data storage. However, built-in globals are sometimes faster for most data types. Another advantage of the built-in globals is that you can have all the global data for your entire program present in just one global variable but access them separately with no penalty in performance. With subVI-based globals, you can combine many variables into one global, but you must read and write them all at once, which increases execution time and memory management overhead. Thus, built-in globals are preferred for most applications for performance reasons.

#### CLAD

A hazard you need to be aware of when using either type of global variable is the potential for **race conditions**. A race condition exists when two or more events can occur in any order, but you rely on them to occur in a *particular* order. While you're developing your VI, or under normal conditions, the order may be as expected and all is well. But under different conditions the order will vary, causing the program to misbehave. Sequence structures and data dependency prevent race conditions from being a general problem in LabVIEW, but global



**Figure 3.13** Race conditions are a hazard associated with all global variables. The global gets written in two places. Which value will it contain when it's read?

variables provide a way to violate strict dataflow programming. Therefore, it's up to you to understand the pitfalls of race conditions.

In Figure 3.13, two While Loops are executing at the same time (they don't necessarily have to be on the same diagram). Both loops write a value to a global number, and one of the loops reads the value. The trouble is, which value will the global contain when it's time to read it? The value from loop A or the value from loop B? Note that there can be any number of data writers out there, adding to the uncertainty. You might have to add an elaborate handshaking, timing, or sequencing scheme to this simple example to force things to occur in a predictable fashion.

An all-around safe approach to avoiding race conditions is to write your overall hierarchy in such a way that a global can only be written from one location. This condition would be met, for instance, by the client-server architecture where there is one data source (a data acquisition VI) with multiple data readers (display, archive, etc.). You must also make sure that you never read from a global before it is initialized.

This is one of the first rules taught for traditional languages: Initialize the variables, then start running the main program. For example, it would be improper for the data display VI to run before the data acquisition VI because the global variable is initially empty or contains garbage. *Rule: Enforce the order of execution in all situations that use global variables.*

If your application has a global array, there is a risk of excessive data duplication. When an array is passed along through wires on a single diagram, LabVIEW does an admirable job of avoiding array duplication, thus saving memory. This is particularly important when you want to access a single element by indexing, adding, or replacing an element. But if the data comes from a global variable, your program has to read the data, make a local copy, index or modify the array, and then write it back. If you do this process at many different locations,

you end up making many copies of the data. This wastes memory and adds execution overhead. A solution is to create a subVI that encapsulates the global, providing whatever access the rest of your program requires. It might have single-element inputs and outputs, addressing the array by element number. In this way, the subVI has the only direct access to the global, guaranteeing that there is only one copy of the data. This implementation is realized automatically when you create functional globals built from shift registers.

Global variables, while handy, can quickly become a programmer's nightmare because they hide the flow of data. For instance, you could write a LabVIEW program where there are several subVIs sitting on a diagram with no wires interconnecting them and no flow control structures. Global variables make this possible: The subVIs all run until a global boolean is set to False, and all the data is passed among the subVIs in global variables. The problem is that nobody can understand what is happening since all data transfers are hidden. Similar things happen in regular programming languages where most of the data is passed in global variables rather than being part of the subroutine calls. This data hiding is not only confusing, but also dangerous. All you have to do is access the wrong item in a global at the wrong time, and things will go nuts. How do you troubleshoot your program when you can't even figure out where the data is coming from or going to?

The answer is another rule. *Rule: Use global variables only where there is no other dataflow alternative.* Using global variables to synchronize or exchange information between parallel loops or top-level VIs is perfectly reasonable if done in moderation. Using globals to carry data from one side of a diagram to the other "because the wires would be too long" is asking for trouble. You are in effect making your data accessible to the entire LabVIEW hierarchy. One more helpful tip with any global variable: Use the Visible Items . . . Label pop-up item (whether it's a built-in or VI-based global) to display its name. When there are many global variables in a program, it's difficult to keep track of which is which, so labeling helps. Finally, if you need to locate all instances of a global variable, you can use the **Find** command from the Edit menu, or you can pop up on the global variable and choose **Find: Global References**.

## Local variables

Another way to manipulate data within the scope of a LabVIEW diagram is with **local variables**. A local variable allows you to read data from or write data to controls and indicators without directly wiring to the usual control or indicator terminal. This means you have unlimited read-write access from multiple locations on the diagram. To create a

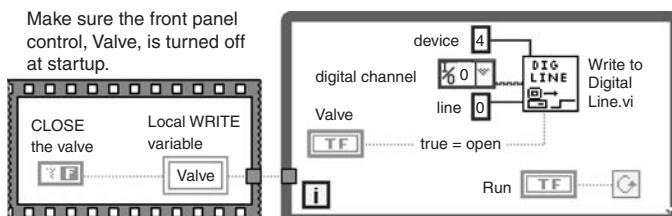
local variable, select a local variable from the Structures palette and drop it on the diagram. Pop up on the local variable node and select the item you wish to access. The list contains the names of every control and indicator on the front panel. Then choose whether you want to read or write data. The local variable behaves exactly the same as the control or indicator's terminal, other than the fact that you are free to read or write. Here are some important facts about local variables.

**CLAD**

- Local variables act only on the controls and indicators that reside on the same diagram. You can't use a local variable to access a control that resides in another VI. Use global variables, or better, regular wired connections to subVIs to transfer data outside of the current diagram.
- You can have as many local variables as you want for each control or indicator. Note how confusing this can become: Imagine your controls changing state mysteriously because you accidentally selected the wrong item in one or more local variables. Danger!
- As with global variables, you should use local variables only when there is no other reasonable dataflow alternative. They bypass the explicit flow of data, obscuring the relationships between data sources (controls) and data sinks (indicators).
- Each instance of a local variable requires an additional copy of the associated data. This can be significant for arrays and other data types that contain large amounts of data. This is another reason that it's better to use wires than local variables whenever possible.

There are three basic uses for local variables: control initialization, control adjustment or interaction, and temporary storage.

Figure 3.14 demonstrates control initialization—a simple, safe, and common use for a local variable. When you start up a top-level VI, it is important that the controls be preset to their required states. In this example, a boolean control opens and closes a valve via a digital

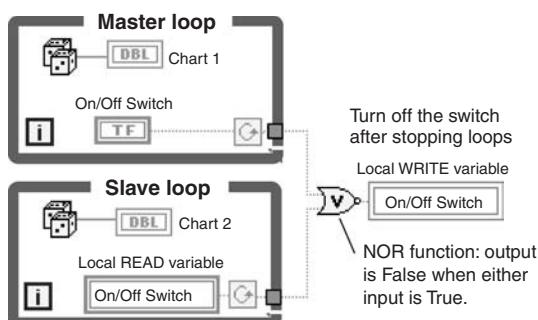


**Figure 3.14** Local variables are a convenient way to initialize front-panel controls. The Sequence structure guarantees that the initialization is completed before the While Loop starts.

output line on a plug-in board. At start-up, a local variable sets the boolean control to False before starting the While Loop. If you have many controls to initialize in this manner, you will need just as many local variables. Note the Sequence structure surrounding the initialization part of the program. This guarantees that the local variable has done its work before the While Loop starts. Otherwise, a race condition might arise.

The same technique is useful at all levels in a hierarchy, but please note that you can initialize the controls in a subVI by assigning them to terminals on the subVI's icon, then wiring in the desired values from the calling VI. This is a case when there is a dataflow alternative. However, if you intend to use a VI at the top level, there is no such alternative, and you can use local variables with our blessing.

Another problem that is solved by local variables is one we have visited before: stopping a parallel While Loop. This is a case where the local variable is a temporary storage device with a scope that extends throughout the diagram of a single VI. In Figure 3.15, we used a local variable to stop the slave loop. There is one catch: You can't use a local variable with a boolean control that has its mechanical action set to one of the *latch* modes. Those are the modes you would normally use for a *stop switch*, where the switch resets to its off position after it is read by the program. Why this limitation? Because there is an ambiguous situation: If the user throws a boolean into its temporary (latched) position, should it reset after being read directly from its regular terminal, or when it is read by a local variable, or both? There is no universal answer to this situation, so it is not allowed. Figure 3.15 has a solution, though. After the two loops stop, another local variable resets the switch for you.



**Figure 3.15** This example shows how to use local variables to stop a parallel While Loop. The switch is programmatically reset because latching modes are not permitted for boolean controls that are also accessed by local variables.

Managing controls that interact is another cool use for local variables. But a word of caution is in order regarding race conditions. It is very easy to write a program that acts in an unpredictable or undesirable manner because there is more than one source for the data displayed in a control or an indicator. What you must do is to explicitly define the order of execution in such a way that the action of a local variable cannot interfere with other data sources, whether they are user inputs, control terminals, or other local variables on the same diagram. It's difficult to give you a more precise description of the potential problems because there are so many situations. Just think carefully before using local variables, and always test your program thoroughly.

To show you how easy it is to create unnerving activity with local variables, another apparently simple example of interactive controls is shown in Figure 3.16. We have two numeric slider controls, for **Potential** (volts) and **Current** (amperes), and we want to limit the product of the two (**Power**, in watts) to a value that our power supply can handle. In this solution, the product is compared with a maximum power limit, and if it's too high, a local variable forces the Potential control to an appropriate value. (We could have limited the Current control as easily.)

This program works, but it has one quirk: If the user sets the Current way up and then drags and holds the Potential slider too high, the local variable keeps resetting the Potential slider to a valid value. However, since the user keeps holding it at the illegal value, the slider oscillates between the legal and illegal values as fast as the loop can run. In the section on events we'll show you an event-driven approach.

**Debugging global variables.** If your program uses many global variables or VIs that use uninitialized shift registers for internal state memory, you have an extra layer of complexity and many more opportunities for

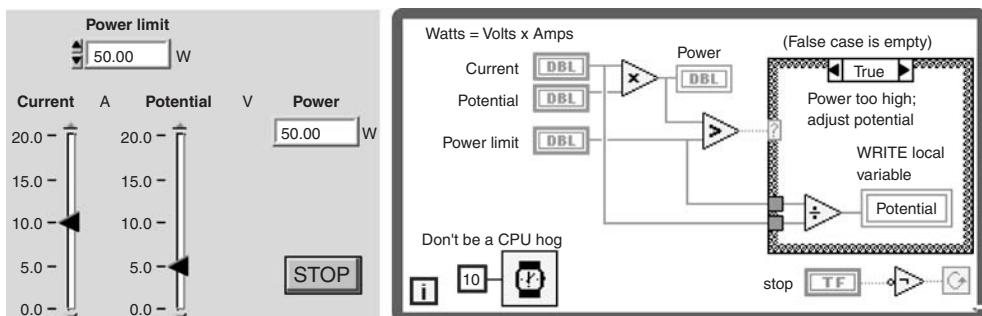


Figure 3.16 Limiting the product of two numeric controls by using local variables. But it acts funny when the user drags and holds the Potential slider to a high value.

bugs to crop up. Here's a brief list of common mistakes associated with global variables and state memory:

- Accidentally writing data to the wrong global variable. It's easy to do; just select the wrong item on one of LabVIEW's built-in globals, and you're in trouble.
- Forgetting to initialize shift-register-based memory. Most of the examples in this book that use uninitialized shift registers have a Case structure inside the loop that writes appropriate data into the shift register at initialization time. If you expect a shift register to be empty each time the program starts, you must initialize it as such. It will automatically be empty when the VI is loaded, but will no longer be empty after the first run.
- Calling a subVI with state memory from multiple locations without making the subVI **reentrant** when required. You need to use reentrancy whenever you want the independent calls *not* to share data, for instance, when computing a running average. Reentrant execution is selected from the VI Properties menu.
- Conversely, setting up a subVI with state memory as reentrant when you *do* want multiple calls to share data. Remember that calls to reentrant VIs from different locations don't share data. For instance, a file management subVI that creates a data file in one location and then writes data to the file in another location might keep the file path in an uninitialized shift register. Such a subVI won't work if it is made reentrant.
- Creating race conditions. You must be absolutely certain that every global variable is accessed in the proper order by its various calling VIs. The most common case occurs when you attempt to write and read a global variable on the same diagram without forcing the execution sequence. Race conditions are absolutely the most difficult bugs to find and are the main reason you should avoid using globals haphazardly.
- Try to avoid situations where a global variable is written in more than one location in the hierarchy. It may be hard to figure out which location supplied the latest value.

How do you debug these global variables? The first thing you must do is to locate all the global variable's callers. Open the global variable's panel and select This VI's Callers from the Browse menu. Note every location where the global is accessed, then audit the list and see if it makes sense. Alternatively, you can use the Find command in the Project menu. Keep the panel of the global open while you run the main VI.

Observe changes in the displayed data if you can. You may want to single-step one or more calling VIs to more carefully observe the data. One trick is to add a new string control called **info** to the global variable. At every location where the global is called to read or write data, add another call that writes a message to the **info** string. It might say, “SubVI abc writing xyz array from inner Case structure.” You can also wire the Call Chain function (in the Advanced function palette) to the **info** string. (Actually, it has to be a string array in this case.) Call Chain returns a string array containing an ordered list of the VI calling chain, from top-level VI on down. Either of these methods makes it abundantly clear who is accessing what.

**Debugging local variables and property nodes.** Everything we've said regarding global variables is true to a great degree for local variables. Race conditions, in particular, can be very difficult to diagnose. *This is reason enough to avoid using local variables except when absolutely necessary!* Execution highlighting and/or single-stepping is probably your best debugging tool because you can usually see in what order the local variables are being executed. However, if your VI has a complicated architecture with parallel loops and timers, the relative ordering of events is time-dependent and cannot be duplicated in the slow motion of execution highlighting. In difficult cases, you must first make sure that you have not inadvertently selected the wrong item in a local variable. Pop up on the problematic control or indicator, and select **Find >> Local Variables**. This will lead you to each local variable associated with that control.

Be sure that each instance is logically correct: Is it read or write mode, and is it the proper place to access this variable? Your only other recourse is to patiently walk through the logic of your diagram, just as you would in any programming language. See? We told you that local variables are a violation of LabVIEW's otherwise strict dataflow concepts. And what did they get you? The same old problems we have with traditional languages.

Property nodes can be a source of bugs just as local variables can. Controls disappear at the wrong time, scales get adjusted to bizarre values, and control sliders jiggle up and down. The pop-up Find command can also locate Property nodes, and you should always go through that exercise as a first step in debugging. Make sure that you have forced the execution sequence for each Property node, and that the sequence makes sense. Complex user interfaces make extensive use of Property nodes, and it's easy to forget to connect an item or two, leaving a control in an inconsistent condition. After a heavy editing session, mistakes are bound to crop up. Again, you must patiently walk through your code.

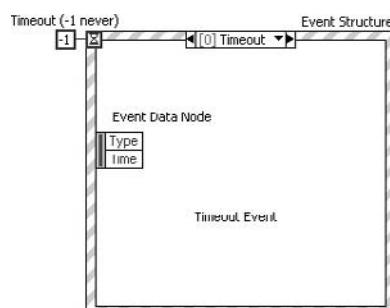
## Events

### CLAD

Asynchronous event-driven applications are fundamentally different from data-flow-driven applications. This conflict between data-flow-driven and event-driven architecture has made writing user-interface event-driven applications in LabVIEW difficult without deviating from dataflow. In the past you had to monitor front panel controls using polling. Figure 3.9 is an example of how polling is used to monitor changes in a front panel control. The value of the numeric control is saved in a shift register and compared with the current value on each iteration of the loop. This was the typical way to poll the user interface. There were other methods, some more complex than others, to save and compare control values; but when you used polling, you always risked missing events. Now with the introduction of the **Event structure** in LabVIEW 6.1, we finally have a mechanism that allows us to respond to asynchronous user-interface events.

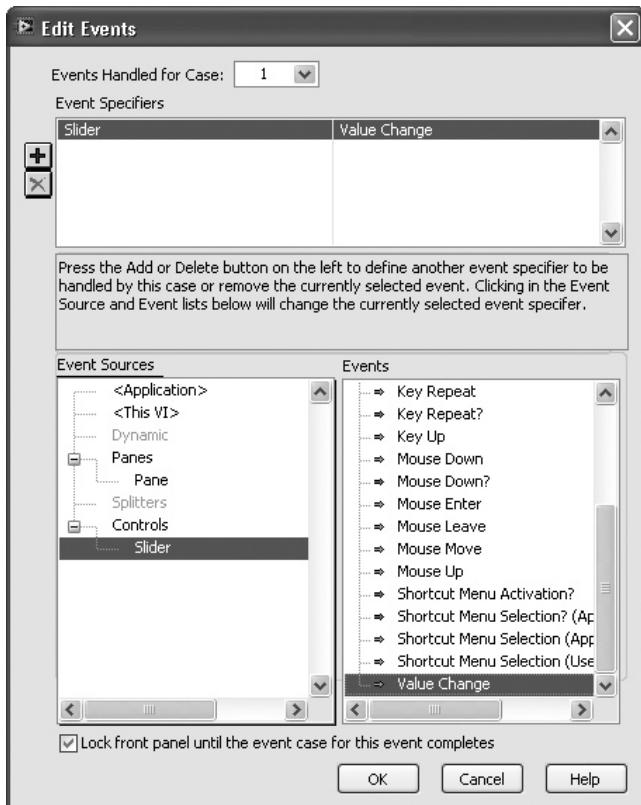
### Notify and Filter events

Let's start off by defining an *event* as a notification that "something" happened. LabVIEW processes event notifications from the operating system, and now with the Event structure we have a way to get in on the action and process events on our block diagram. The Event structure is shown in Figure 3.17. You'll find it on the Programming >> Structures Function palette. It looks a lot like a Case structure with multiple cases and a selector on top. You can configure it to fire on **Application events** (that is, application exiting, etc.), **VI events** (panel close, window move, etc.), **Sub Panel Pane events**



**Figure 3.17** The Event structure looks a lot like a Case structure, with different event cases configured to fire on specific events. The Event structure sleeps until an event fires or the timeout expires.

(mouse down, mouse move, etc.), **Control and Indicator events** (mouse down, mouse move, key down, etc.); you can even configure your own **Dynamic** user-defined events. Only one case fires for each event—the structure does not do any looping on its own. The Event structure has a time-out value in milliseconds. The default is -1, or never time out. While it's waiting, the Event structure sleeps and consumes no CPU cycles. Inside each event is a data node that returns information about the event such as the time the event fired. Need to log every keystroke and when it occurred? Want to add a macro capability to your application? Events make it easy. In Figure 3.18 we're adding an Event case to register for event notifications from the slider control on the front panel. Once registered for the Value Change event, this case will fire every time the slider changes value. The Edit Events dialog shows just a few of the events we can register for

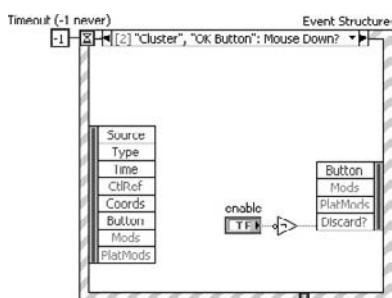


**Figure 3.18** Register for the Value Change event from the Slider control. This event case will fire every time the slider changes value.

this one control. The Value Change event is classified as a **Notify** event. For the Value Change Notify event, LabVIEW processes the user-interface action (changing the control value) and then notifies all the Event structures waiting for the event that the value has changed. We could have multiple Event structures (not generally a good idea) waiting for notification of this event, and LabVIEW would notify them all in parallel.

LabVIEW passes **Filter** events (Filter events end with a “?”) to you before processing so you can decide how to handle the event. For example, the event in Figure 3.19 is configured to filter Mouse Down? events for two controls: Cluster and OK button. While the Enable button is False, mouse down actions on these two controls will be discarded. Filter events let you decide what to do with the users’ action. Note that we are using a single event case to wait on events from multiple controls. We can do this with Filter events if the events return identical event data. Notify events can be merged, but the left data node will return only the data common to all events. In the section on dynamic events we’ll see this again.

Notify and Filter events open up a whole new world of user-interface interaction previously impossible with polling. This newfound power also requires us to learn some new ways to program. If we’re not careful, the Event structure has some characteristics that can jump up and bite us. In Figure 3.18 the Edit Events dialog box has a check box labeled “Lock front panel until the event case for this event completes.” And that does exactly what it says: When the Value Change event for the slider fires, the front panel is locked until the event case completes operation.

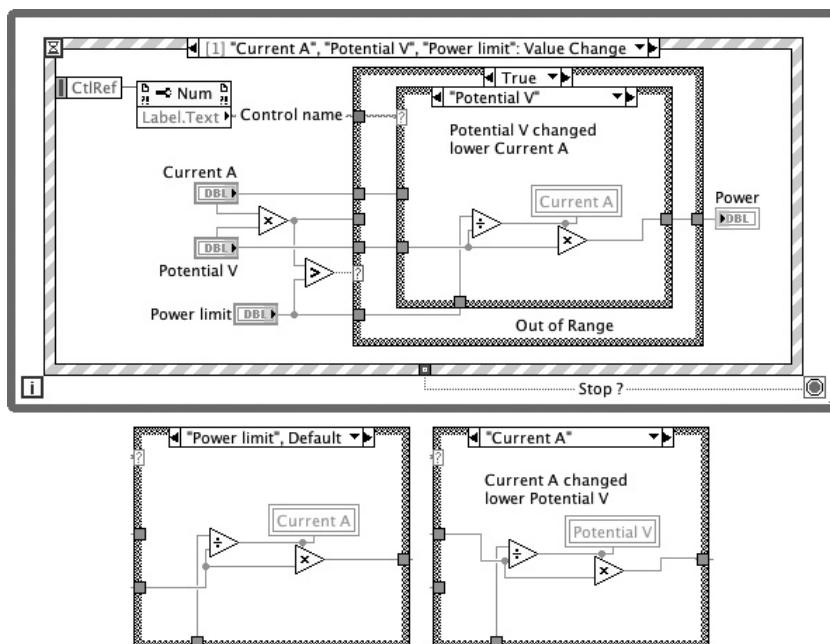


**Figure 3.19** Filter events allow you to decide whether to let the event happen. While the “Enable” button is False, the mouse down events on the Cluster and OK buttons are discarded. Use Filter events when you want to intercept and process events before the event is handled by LabVIEW.

The locking action gives you a chance to swap out the user interface or whatever else you need to do to handle the event. The user will not notice anything as long as you process the event and quickly exit the event case. However, if you place code in your event case that takes a while to execute, or if your Event structure is buried in a state machine, the front panel of your VI will be totally unresponsive. Some practical guidelines are as follows:

- Use only one Event structure per block diagram. Let a single Event structure provide a focal point for all user-interface code.
- Do only minimal processing inside an Event structure. Lengthy processing inside the event case will lock up the user interface.
- Use a separate While Loop containing only the Event structure. In the section on design patterns we'll show you a good parallel architecture for user-interface-driven applications.

You can combine events, Property nodes, and local variables to create event-driven applications that are extremely responsive. Because the Event structure consumes no CPU cycles while waiting for an event, they make a very low overhead way to enhance your user interface. Figure 3.20 shows the application from Figure 3.16 rewritten using an



**Figure 3.20** Limiting the product of two numeric controls using local variables inside an Event structure. Interaction is smooth and easy compared to that in Figure 3.16.

Event structure. We use a single event case to fire on a Value Change event from any of the three controls. The left data node of the event case returns a reference to the control that triggered the event from which we can get the name using a Property node. If the calculated power exceeds the power limit, we adjust the opposite control's value. The result is a responsive piece of user-interface code that works seamlessly and consumes almost no CPU cycles, unlike earlier examples that used polling.

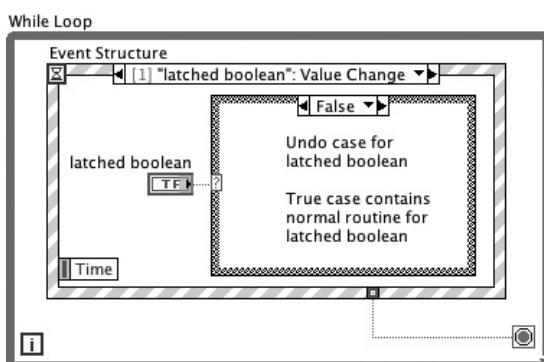
## Mechanical actions

The cleanest, easiest, and recommended way to use events and stay within the dataflow concept is to put controls inside their event case. This is especially important for boolean controls with a latching mechanical action. To return to its default state, a boolean latch depends on being read by the block diagram. If you place the latching boolean inside its event case, the mechanical action will function properly. But there's a funny thing about Value Change events and latched booleans. What happens when the user selects Edit >> Undo Data Change? The event fires again, but the control value is not what you expected. Figure 3.21 illustrates how to handle this unexpected event by using a Case structure inside the event for any latched boolean. The True case handles the normal value change, and the False case handles the unexpected Undo.

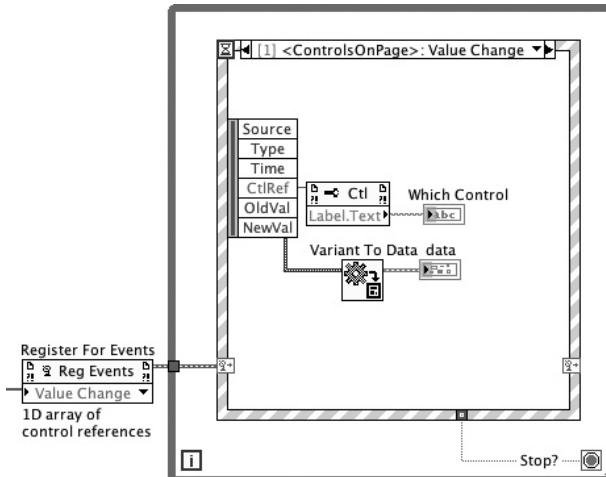
## Dynamic events

### CLAD

Dynamic events allow you to register and unregister for events while your application is running. The Filter and Notify events shown so far were statically linked at edit time to a specific control. As long as the VI

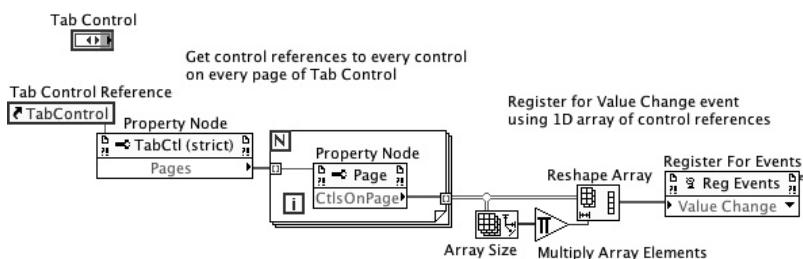


**Figure 3.21** Value Change events for booleans with a latched mechanical action require special attention.



**Figure 3.22** Dynamic events programmatically register and unregister controls for events. Note the data from the controls is returned as a variant. Dynamic events are not limited to front panel controls, but can be programmatically generated.

is running, changes to the control will fire a Value Change event. Usually this is the behavior you want. But there are some situations where dynamic events are a better solution. In Figure 3.22 we used control references to dynamically register controls for a Value Change event. The Register For Events property node receives an array of references of all the controls on every page of a tab control (Figure 3.23). Dynamically registering for this event can save a lot of development time in high-channel-count applications. Note the data from the control is returned as a variant. In our case all the controls were clusters with the same data type, so it is simple to extract the data from the variant. For clarity the actual controls were not placed in the Value Change



**Figure 3.23** A reference to Tab Control is used to register every control on every page of Tab Control for the Value Change event. This could easily be turned into a reusable user-interface subVI.

event case, but good programming practice suggests you should put them there.

Dynamically registering controls for a Value Change event only touches the tip of the iceberg in what you can do with dynamic events. You can create your own user-defined events and use the Event structure as a sophisticated notification and messaging system. You can also use the Event structure for ".NET" events. Whatever you use the Event structure for, try not to stray too far from dataflow programming. In the section on design patterns we'll go over some proven ways to use the Event structure in your applications.

## Design Patterns

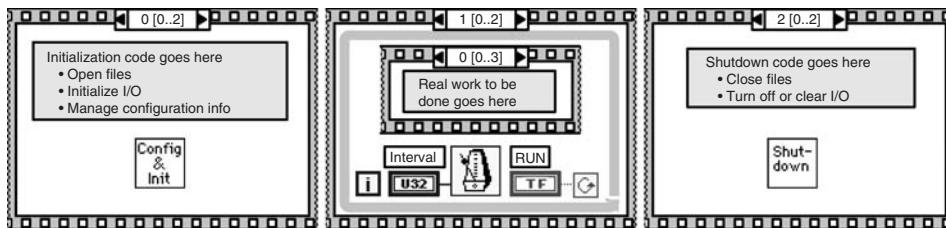
Your initial objective is to decide on an overall **architecture**, or programming strategy, which determines how you want your application to work in the broadest sense. Here are some models, also called LabVIEW **Design Patterns**, that represent the fundamental structure of common LabVIEW applications. The goals of using design patterns are that the architecture has already been proved to work, and when you see someone else's application that incorporates a design pattern you've used before, you recognize and understand it. The list of design patterns continues to grow as the years go by, and they are all reliable approaches. Choosing an appropriate architecture is a mix of analysis, intuition, and experience. Study these architectures, memorize their basic attributes, and see if your next application doesn't go together a little more smoothly.

### Initialize and then loop

The simplest applications initialize and then loop. You perform some initialization, such as setting up files and starting the hardware, and then drop into a While Loop that does the main task several times. The main loop usually consists of four steps that are repeated at a rate that matches the requirements of your signals:

1. Acquire.
2. Analyze.
3. Display.
4. Store.

In a well-written program, these steps would be encapsulated in subVIs. After the main loop has terminated, there may also be a shutdown or cleanup task that closes files or turns off something in the hardware.

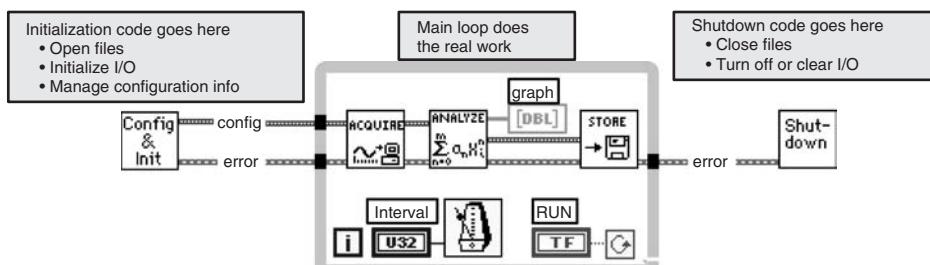


**Figure 3.24** A Sequence containing frames for starting up, doing the work, and shutting down. It is a classic for data acquisition programs. The While Loop runs until the operation is finished. The Sequence inside contains the real work to be accomplished.

This strategy is a classic for the top-level VI in ordinary data acquisition. The first two design patterns, which follow, work this way.

Figure 3.24 is by far the most commonly used program structure in all of LabVIEW, primarily because it is an obvious solution. An overall Sequence structure forces the order of initialize, main operation, and shutdown. The main While Loop runs at a regular interval until the user stops the operation or some other event occurs. Frequently found inside the main loop is another Sequence structure that contains the operations to be performed: acquire, analyze, display, and then store. Unfortunately, this approach obscures dataflow with all those Sequence structures.

An improvement on the previous example, shown in Figure 3.25, makes effective use of dataflow programming and can be somewhat easier to understand. Connections between the three major operations (initialize, main operation, and shutdown) force the order of execution in the same way that an overall Sequence structure does, except there is no obscuring of the overall program. The same methodology can and should be applied inside the main While Loop by using common threads between the functions employed there.



**Figure 3.25** Dataflow instead of Sequence structure enhances readability. It has the same functionality as Figure 3.24. Note that the connections between tasks (subVIs) are not optional: they force the order of execution.

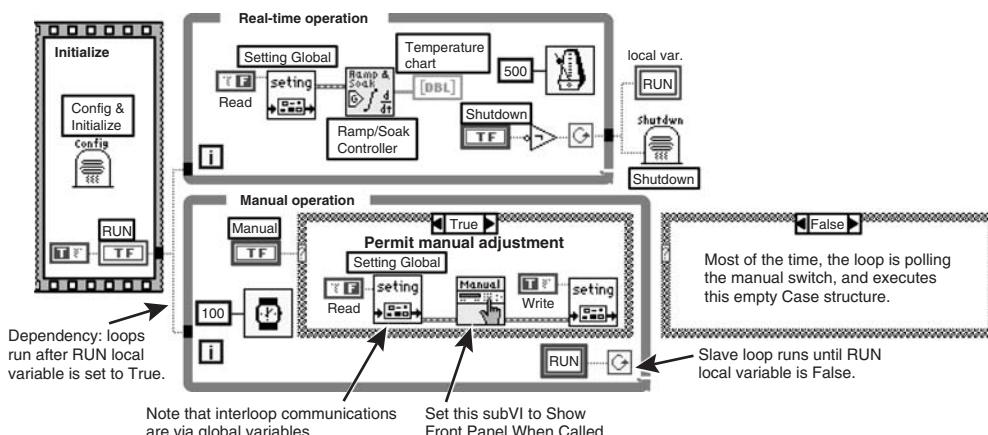
The net result is a clearer program, explainable with just one page. This concept is used by most advanced LabVIEW users, like you. The example shown here is in every way a real, working LabVIEW program.

### Independent parallel loops

Any time you have two tasks that you need to do at different speeds or priorities, consider using multiple independent While Loops that operate in parallel on one diagram (Figure 3.26). You can also use independent loops when there is a main task that needs to run regularly and one or more secondary tasks that run only when the operator throws a switch. If data needs to be exchanged between the loops, as it often does, use local or global variables. In this example, a boolean control with a local variable stops the loops at the appropriate time, and a smart global variable (operating as a little database) carries a cluster of settings between the two loops. This is another top-level VI architecture.

Remember to initialize controls before they are read anywhere else. If you are using a loop similar to the bottom one shown in this example, put a time delay inside the loop to keep it from hogging CPU time when nothing is happening. Normally we'll use 100 or 200 ms because that's about the threshold of human perception.

Global variables tend to hide the flow of data if they are buried inside subVIs. To make your programs easier to understand, a good policy is to place all global variables out on the diagram where they are

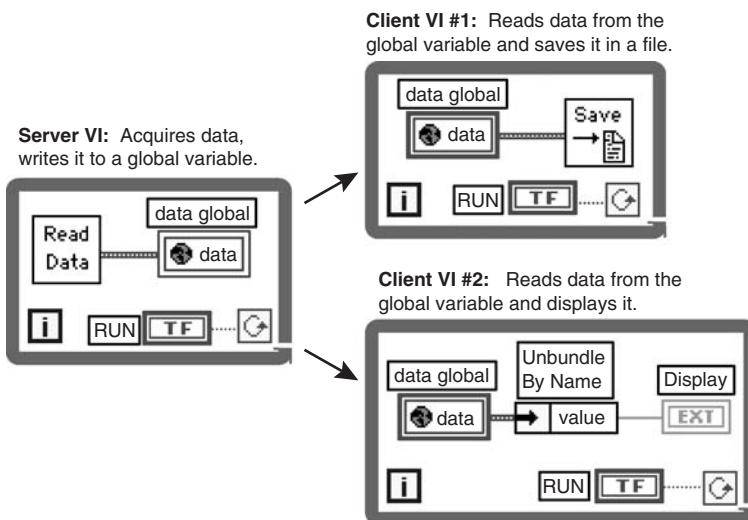


**Figure 3.26** This example uses two independent loops: the upper one for the actual control and the lower one to handle manual operation (shown simplified here). Thus the main control loop always runs at a constant interval regardless of the state of the manual loop.

easily spotted. In this example, the Manual Control subVI reads settings information from a global, changes that information, and then writes it back to the global. We could have hidden the global operations inside the subVI, but then you wouldn't have a clue as to how that subVI acts on the settings information, which is also used by the Ramp/Soak Controller subVI. It also helps to show the name of the global variable so that the reader knows where the data is being stored. We also want to point out what an incredibly powerful, yet simple technique this is. Go ahead: Implement parallel tasks in C or Pascal.

### Client-server

The term ***client-server*** comes from the world of distributed computing where a central host has all the disk drives and shared resources (the **server**), and a number of users out on the network access those resources as required (the **clients**). Here is how it works in a LabVIEW application: You write a server VI that is solely responsible for, say, acquiring data from the hardware. Acquired data is prepared and then written to one or more global variables that act as a database. Then you design one or more client VIs that read the data in the global variable(s), as shown in Figure 3.27. This is a very powerful concept, one that you will see throughout this book and in many advanced examples.



**Figure 3.27** Client-server systems. Client VIs receive data from a server VI through the use of a global variable. This permits time independence between several VIs or loops.

A good use for this might be in process control (Chapter 18, “Process Control Applications”) where the client tasks are such things as alarm generation, real-time trending, historical trending to disk, and several different operator-interface VIs. The beauty of the client-server concept is that the clients can be

- Independent loops on one diagram
- Completely independent top-level VIs in different windows
- Running at different rates
- Running on different machines if you use a network

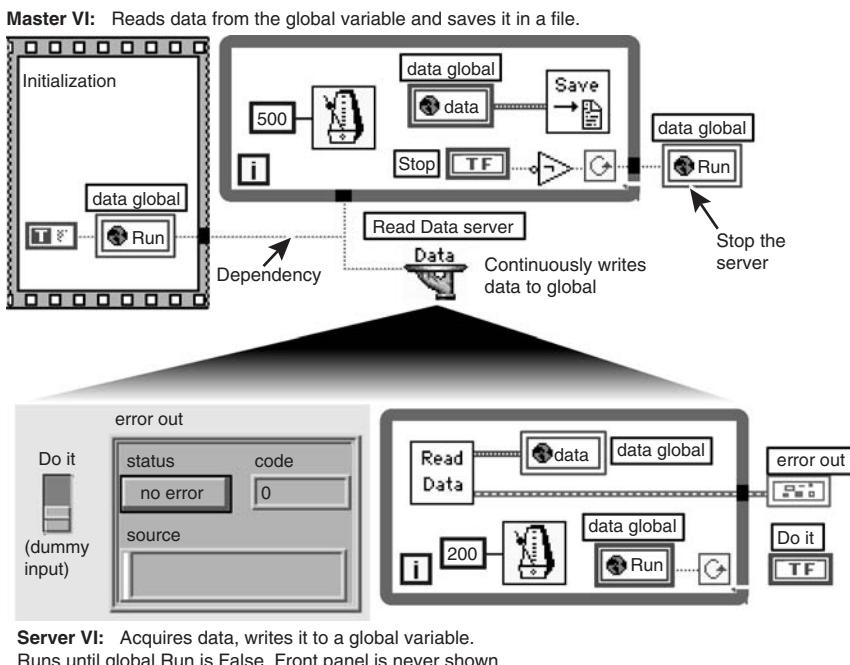
If you want one button to shut down everything, add a RUN boolean control to a global variable and wire that into all the clients. You can also set the **priority** of execution for each VI by two different means. First, you can put a timer in each loop and make the high-priority loops run more often. This is simple and foolproof and is generally the recommended technique. Second, you can adjust the execution priority through the VI Properties dialog. Typically, the server would have slightly higher priority to guarantee fresh data and good response to the I/O systems while the clients would run at lower priority. Use execution priority with caution and a good dose of understanding about how LabVIEW schedules VI execution. In a nutshell, high-priority VIs cut in line ahead of low-priority VIs. If a high-priority VI needs service too often, it may starve all the other VIs, and your whole hierarchy will grind to a halt. You can read more about this topic in the LabVIEW User’s Manual, in the chapter on performance issues. Other references on this topic are listed in the Bibliography.

**Race conditions** are a risk with the client-server architecture because global variables may be accessed without explicit ordering. The most hazardous situation occurs when you have two or more locations where data is *written* to a global variable. Which location wrote the latest data? Do you care? If so, you must enforce execution order by some means. Another kind of race condition surrounds the issue of data aging. For example, in Figure 3.27, the first client VI periodically stores data in a file. How does that client know that fresh data is available? Can it store the same data more than once, or can it miss new data? This may be a problem. The best way to synchronize clients and servers is by using a global **queue**. A *queue* is like people waiting in line. The server puts new data in the back of the queue (data is *enqueued*), and the clients dequeue the data from the front at a later time. If the queue is empty, the client must wait until there is new data. For a good example of the queue technique, see the LabVIEW examples in the directory examples/general/synchexm.llb/QueueExample.VI.

### Client-server (with autonomous VIs)

Autonomous client or server VIs can be launched by a master VI to run independently without being placed in a While Loop on the diagram of the master VI. Figure 3.28 shows how. Operation of the autonomous Read Data Server VI is quite similar to that of the previous example in that it acquires data at an independent rate and transmits it to the master (client) VI through a global variable or queue. The server VI is stopped by a global boolean when commanded by the master VI. Note that the server VI is placed *outside* the While Loop in the master VI because the server is called only once and is intended to run until the master VI terminates. If the server were placed *inside* the While Loop, that loop would not even finish its first cycle; it would wait (forever) until the server VI finished.

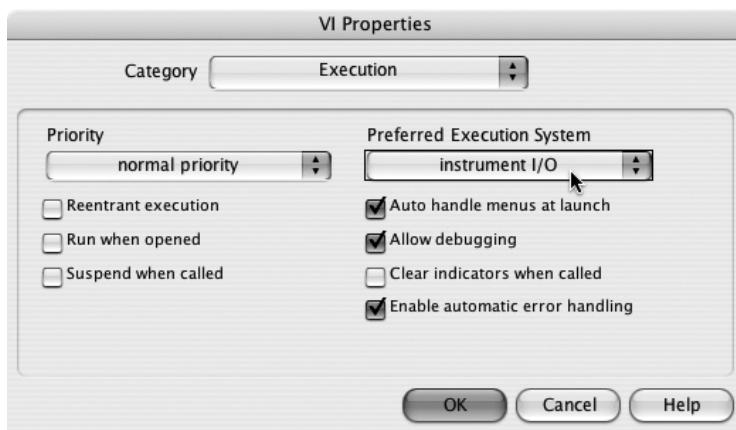
There can be multiple autonomous VIs, and their panels may or may not appear when the VIs are called. For instance, an alarm display VI would certainly be displayed, while a VI that simply logs data to disk probably would not. What if an error occurs in the autonomous VI? You have several options. You can do nothing, as in my simple example, or maybe return an error cluster when execution is finished. You could



**Figure 3.28** The upper diagram is a master VI (and client) that launches the lower VI, which is an autonomous server.

have the VI generate a dialog box, but what happens after that? A versatile solution is to have the autonomous VI set the Run global boolean to False and wire the master VI in such a way that it is forced to stop when the Run global becomes False. That way, any VI in the hierarchy could potentially stop everything. Be careful. Don't have the application come to a screeching halt without telling the user *why* it just halted.

There is only a slight advantage to the autonomous VI solution compared to placing multiple While Loops on the main diagram. In fact, it's an obscure performance benefit that you'll notice only if you're trying to get the last ounce of performance out of your system. Here's the background: LabVIEW runs sections of code along with a copy of the execution system in threads. This is called multithreading. The operating system manages when and where the threads run. Normally a subVI uses the execution system of its calling VI, but in the VI Properties dialog (Figure 3.29), you can select a different execution system and therefore ensure a different thread from the calling VI. Running a subVI in a separate execution system increases the chances that it will run in parallel with code in other execution systems on a multi-processor system. Normally the disadvantage to placing a subVI in a different execution system is that each call to the subVI will cause a context switch as the operating system changes from one execution system (thread) to another. That's the background; now for the case of the autonomous VI. If the autonomous VI is placed in a separate execution system, there will only be one context switch when it is originally called, and then the autonomous VI will be merrily on its way in its own execution system. As we said, the advantage is slight, but it's there if you need it.



**Figure 3.29** There is a slight performance benefit to placing autonomous VIs in their own execution system.

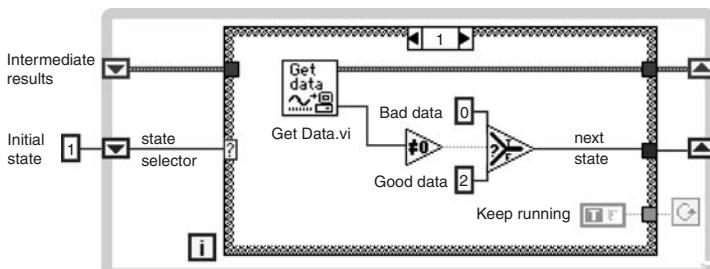
## State machines

### CLAD

There is a very powerful and versatile alternative to the Sequence structure, called a **state machine**, as described in the advanced LabVIEW training course. The general concept of a state machine originates in the world of digital (boolean) logic design where it is a formal method of system design. In LabVIEW, a state machine uses a Case structure wired to a counter that's maintained in a shift register in a While Loop. This technique allows you to jump around in the sequence by manipulating the counter. For instance, any frame can jump directly to an error-handling frame. This technique is widely used in drivers, sequencers, and complex user interfaces and is also applicable to situations that require extensive error checking. Any time you have a chain of events where one operation depends on the status of a previous operation, or where there are many modes of operation, a state machine is a good way to do the job.

Figure 3.30 is an example of the basic structure of a state machine. The **state number** is maintained as a numeric value in a shift register, so any frame can jump to any other frame. Shift registers or local variables must also be used for any data that needs to be passed between frames, such as the results of the subVI in this example. One of the configuration tricks you will want to remember is to use frame 0 for errors. That way, if you add a frame later on, the error frame number doesn't change; otherwise, you would have to edit every frame to update the error frame's new address.

Each frame that contains an activity may also check for errors (or some other condition) to see where the program should go next. In driver VIs, you commonly have to respond to a variety of error



**Figure 3.30** A basic state machine. Any frame of the case can jump to any other by manipulating the state selector shift register. Results are passed from one frame to another in another shift register. Normal execution proceeds from frame 1 through frame 2. Frame 0 or frame 2 can stop execution. This is an exceptionally versatile concept, well worth studying.

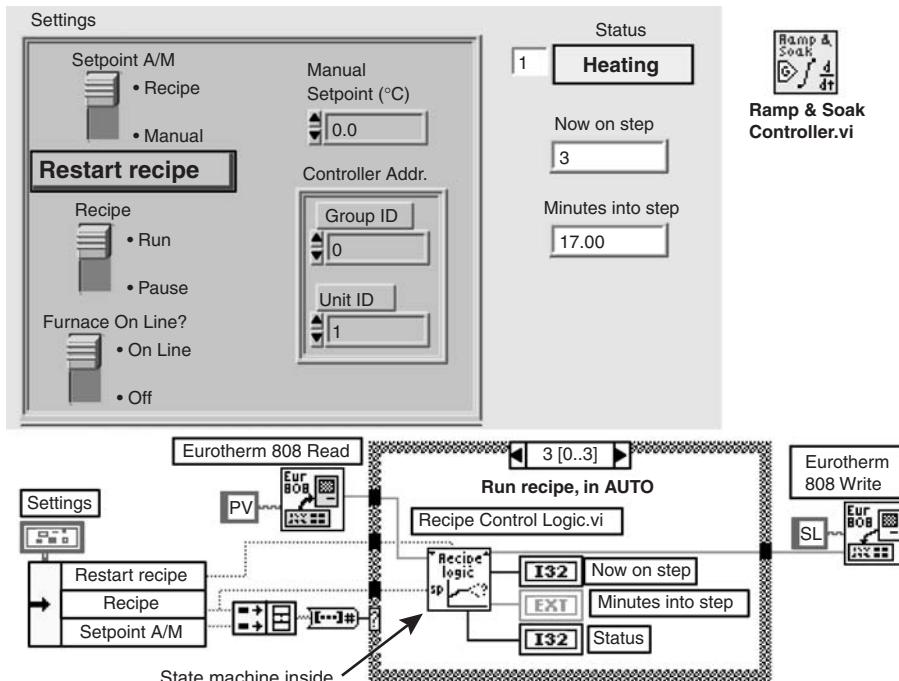
conditions where you may want to retry communications, abort, or do something else depending on the condition detected. Similarly, in a complex user interface, you may have to handle major changes in display configuration (using Property nodes) or handle errors arising from user data entries. The state machine lets you handle these complex situations. In this example, the program starts in frame 1 where data is acquired. If the data is no good, the program jumps to frame 0, reports the problem, and stops the While Loop. If the data is OK, it gets analyzed in frame 2. Note that an infinite loop could occur here; additional conditions should be added in a practical program to guarantee stopping after some number of retries.

*Tip:* It's easy to forget to stop the While Loop after the final frame. The program will execute the last frame over and over, often with very strange results. Remember to put the little False boolean constant where it belongs—in the error handler frame *and* in the last frame of the sequence.

A state machine can be the diagram behind a flexible user interface, or it can be called as a subVI. In a user interface, the state machine manages various modes of operation. For instance, the user may have a series of mode selections when configuring an experiment. A mode selector switch may directly drive the Case structure, or you may use Property nodes to selectively hide and show valid mode selection buttons.

Depending upon which button is then pressed, the state machine jumps to an appropriate frame to handle that selection.

A favorite use for state machines is intelligent subVIs in such applications as sequencers and profile generators. In those situations, the state machine subVI is called periodically in the main While Loop. Each time it's called, information from the last cycle is available in the shift register, providing the required history: Where have I been? Data passed in through the connector pane provides the current information: Where am I now? And computation within the state machine subVI determines the future: Where should I go next? Those values may drive an I/O device through a subVI within the state machine, or they may be passed back to the calling VI through the connector pane. Figure 3.31 is an example, and it works as just described. This VI is called every second from the main While Loop. Inside, it calls a state machine, Recipe Control Logic, which keeps track of the state of a complex ramp-and-soak temperature recipe. On each iteration, the state machine receives the current process variable (temperature) measurement and some commands regarding whether or not to continue running the recipe. It computes a new controller set point, which is returned and then transmitted to the Eurotherm 808 controller.



**Figure 3.31** A peek inside the Ramp-and-Soak Controller VI (simplified), which is called from the main VI. A state machine, Recipe Control Logic, keeps track of complex temperature profile recipes.

Great confusion can arise when you are developing a state machine. Imagine how quickly the number of frames can accumulate in a situation with many logical paths. Then imagine what happens when one of those numeric constants that points to the next frame has the *wrong number*. Or even worse, what if you have about 17 frames and you need to add one in the middle? You may have to *edit* dozens of little numeric constants. And just wait until you have to explain to a novice user how it all works!

Thankfully, creative LabVIEW developers have come up with an easy solution to these complex situations: Use **enumerated constants** (also called *enums*) instead of numbers. Take a look at Figure 3.32, showing an improved version of our previous state machine example. You can see the enumerated constant, with three possible values (error, get data, calculate), has replaced the more cryptic numeric constants. The Case structure now tells you in English which frame you're on, and the enums more clearly state where the program will go next. Strings are also useful as a self-documenting case selector.

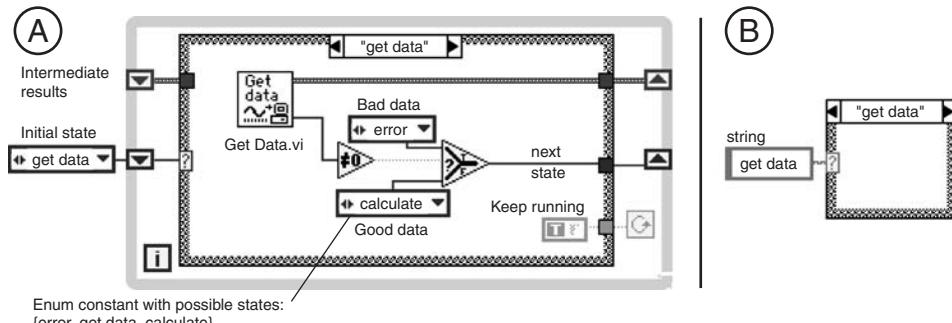


Figure 3.32 (A) A recommended way to write state machines, using enumerated constants. (B) Strings are also useful as self-documenting selectors.

Even with enums or strings as selectors, there's plenty of room for confusion in designing, debugging, or modifying state machines. Life is a bit easier when you design them with the aid of a **state diagram**, an old digital logic designer's tool shown in Figure 3.33. Each bubble represents a state where a task is performed and corresponds to a frame in the Case structure. Paths from one state to another are annotated to show why the transition is made. Many computer-aided software engineering (CASE) applications have a state diagram tool

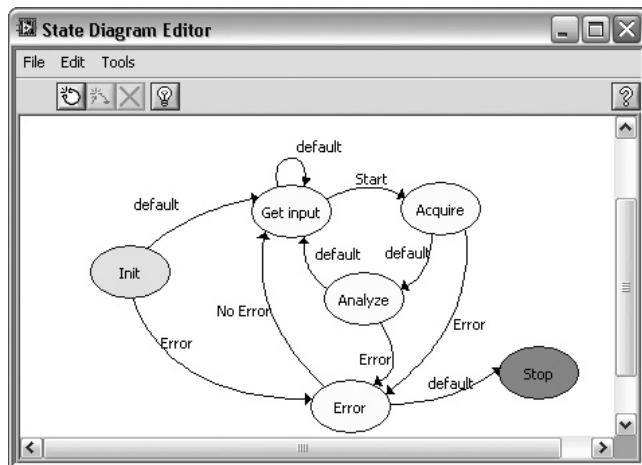
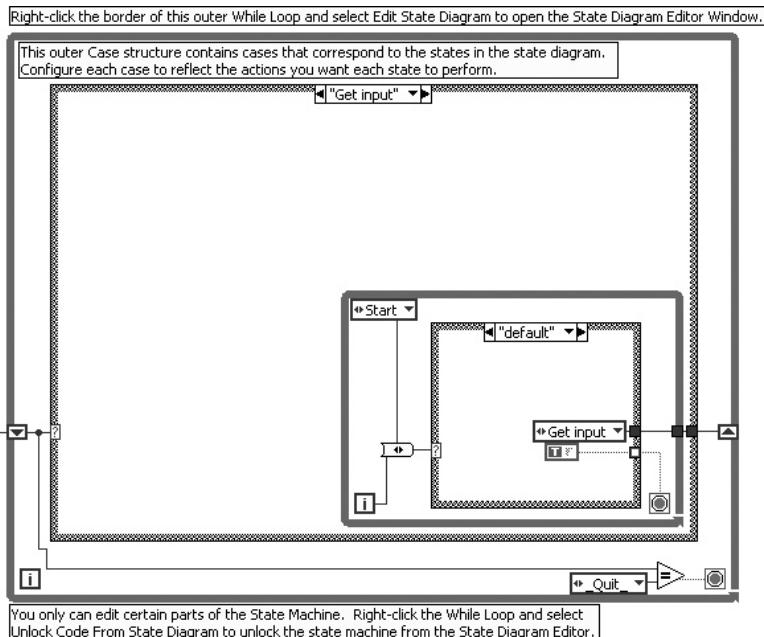


Figure 3.33 LabVIEW State Diagram Editor makes it easy to design and modify complex state machines. Notice that the only way out of this state machine is through an error – whoops! Another transition needs to go from Get Input to Stop.



**Figure 3.34** LabVIEW state machine created with the State Diagram Editor Toolkit. It is easy to design and prototype, but you cannot modify the state machine without the editor.

with additional features to simplify state machine design. Figure 3.33 is from the LabVIEW State Diagram Editor Toolkit, and Figure 3.34 shows the block diagram it creates.

#### Queued message handler

A **Queued Message Handler** takes a list (queue) of messages and processes them in order. Originally called queued state machines, they are actually fundamentally different from a state machine. A state machine has predefined transitions built into its architecture. In the state diagram of Figure 3.33 there is a flaw in the logic built into the state machine, and you cannot get to the stop case without going through the error case. Because the behavior is built in, you don't have a way around it without editing the state machine. State machines are best when you have a strictly defined behavior (and you code it properly). That's often not the case in the lab; the equipment may stay the same, but the experiment changes from day to day. When the components of your program stay the same but the order of execution changes, a queued message handler is a powerful tool.

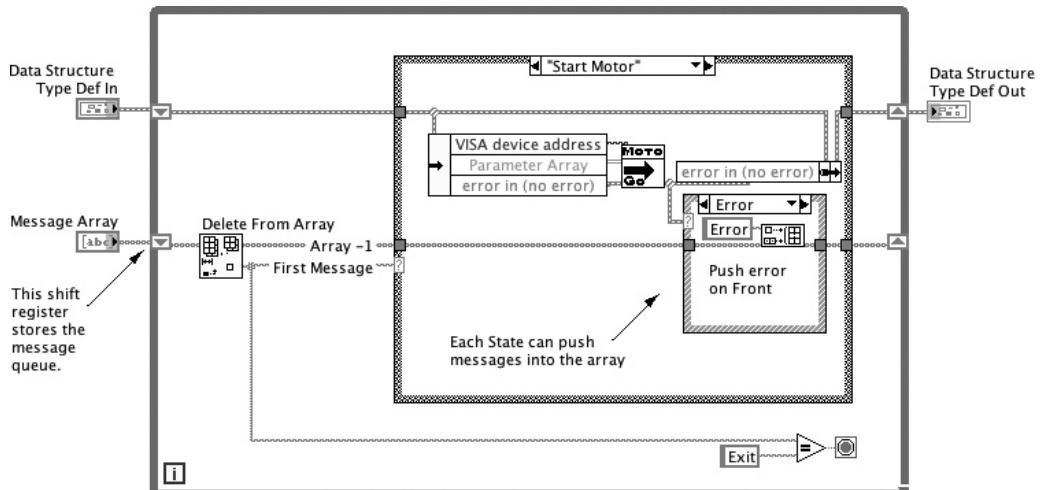


Figure 3.35 A queued message handler sequences through a list of messages.

Each case of a queued message handler can jump to any other case, but unlike for a state machine, the order of the execution is defined by the list of commands (message queue or array) that you pass in. In Figure 3.35 the message queue is held in a shift register. With each iteration of the While Loop, a message at the front of the queue is pulled off and the appropriate case executes. The loop terminates when the queue is empty or an Exit command is reached. Note that we're also using a type defined cluster as a Data structure. The Data structure is a defined mechanism providing operating parameters for subVIs and a way to get data in and out of the queued message handler. The VI in Figure 3.35 would be used as a subVI in a larger application.

The queued message handler architecture works best in combination with other design patterns. It's tempting to create a queued message handler with many atomic states, giving you fine-grained control over your equipment, but this creates problems when one state depends on another state which depends on another state, etc. Then the calling VI has to know too much about the inner workings of the queued message handler. A better solution is to use the queued message handler to call intelligent state machines. Each intelligent state machine manages all the specifics of a piece of equipment or of a process. Now the queued message handler specifies actions, not procedures. Once you master this, you are well on your way to building robust, reusable applications that are driven by input parameters, not constrained by predefined behavior.

## Event-driven applications



Event-driven applications are almost trivially simple with the Event structure. Our guidelines for using the Event structure are as follows:

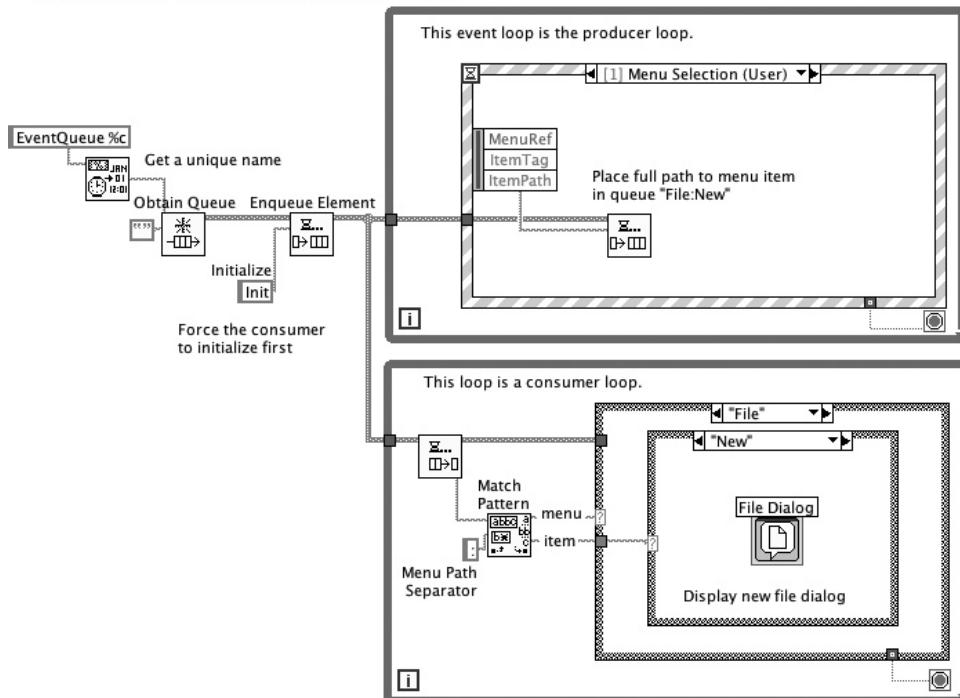
- Use only one Event structure per block diagram. Let a single Event structure provide a focal point for all user-interface code.
- Do only minimal processing inside an Event structure. Lengthy processing inside the event case will lock up the user interface.
- Use a separate While Loop containing only the Event structure.

The locking nature of the Event structure and the need for immediate attention to events dictate that you use two parallel loops. You can use a single loop for both the Event structure and the rest of your code, but eventually you'll run up against a timing issue requiring you to parallelize your application. The Producer/Consumer Design Pattern template in LabVIEW provides a great place to start.

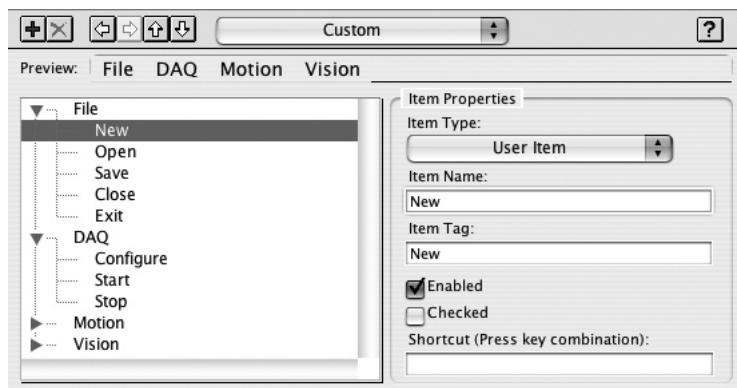
Figure 3.36 is a modified version of LabVIEW's Producer/Consumer Design Pattern. The consumer loop is very similar to the queued message handler. Instead of passing in an array we're using the queue functions to pass in each item as it happens. A queue is a FIFO (first in, first out). Messages are *enqueued* at one end and *dequeued* at the other. For more on queues and other synchronization functions, look at Chapter 6, "Synchronization." Note that we're giving our queue a unique name based on the time of creation. Queues are referenced by name, and all you need is the name of the queue to put items in it. If we left the name empty, there's a real good chance another nameless queue would put items in ours with disastrous results. We're also using a string as the queue element, because they're easy to use as case selectors. After creation the first thing we do is to enqueue "Init," telling our consumer loop to run its initialization case. In the earlier design patterns, initialization routines were outside the main loop. We still initialize first, but keep our block diagram clean and simple. We can also reinitialize without leaving our consumer loop.

The event structure is the heart of the producer loop. The event case in Figure 3.36 is registered for user menu events. Select "Edit >> Run-Time Menu..." to create a custom menu for your application (Figure 3.37). As soon as the user makes a menu selection from your running application, the item path is enqueued in the event case and dequeued in the consumer loop. The full item path is a string with a colon separator between items. Passing menu items as the full item path eliminates errors we might have if two menus had an item with the same name. When the user selects New from the File menu, we get the string "File:New." The consumer loop uses **Match Pattern** to split

This template is from the Producer/Consumer design pattern.



**Figure 3.36** Producer/Consumer Design Pattern modified for event-driven application. User menu selections are placed in the queue and handled in the consumer loop. Match Pattern splits the menu item path at the ":" (colon) separator. Nested cases handle the event actions. (Error handling is omitted for clarity.)



**Figure 3.37** Create a custom menu for your application and use the event-driven Producer/Consumer Design Pattern to handle user selections.

the string at the first colon. We get the two strings File and New. If there is not a colon in the string, Match Pattern returns the whole string at its “before substring” terminal. Passing the full item path between loops also allows us to modularize the consumer loop with a case hierarchy based partly on menu structure. This event-driven design pattern is powerful and very easy to design.

Design patterns provide a starting point for successful programming. There is no one and only design pattern; otherwise, we would have shown you that first and omitted the rest. Pick the design pattern that best fits your skill level and your application. There is no sense in going for something complicated if all you need is to grab a few measurements and write them to file.

## Bibliography

- AN 114, *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability*, www.ni.com, National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2000.
- AN 199, *LabVIEW and Hyper-Threading*, www.ni.com, National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2004.
- Brunzie, Ted J.: “Aging Gracefully: Writing Software That Takes Changes in Stride,” *LabVIEW Technical Resource*, vol. 3, no. 4, Fall 1995.
- “Inside LabVIEW,” NIWeek 2000, Advanced Track 1C, National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2000.

## LabVIEW Data Types

Our next project is to consider all the major data types that LabVIEW supports. The list consists of **scalar** (single-element) types such as numerics and booleans and **structured** types (containing more than one element) such as strings, arrays, and clusters. The LabVIEW *Controls* palette is roughly organized around data types, gathering similar types into subpalettes for easy access. There are, in fact, many ways to *present* a given data type, and that's the main reason that there are apparently so many items in those palettes. For instance, a given numeric type can be displayed as a simple number, a bar graph, a slider, or an analog meter, or in a chart. But underneath is a well-defined representation of the data that you, the programmer, and the machine must mutually understand and agree upon as part of the programming process.

**CLAD**

One thing that demonstrates that LabVIEW is a complete programming language is its support for essentially all data types. Numbers can be floating point or integer, with various degrees of precision. Booleans, bytes, strings, and numerics can be combined freely into various structures, giving you total freedom to make the data type suit the problem. **Polymorphism** is an important feature of LabVIEW that simplifies this potentially complicated world of data types into something that even the novice can manage without much study. Polymorphism is the ability to adjust to input data of different types. Most built-in LabVIEW functions are polymorphic. Ordinary virtual instruments (VIs) that you write are not truly polymorphic—they can adapt between numeric types, but not between other data types such as strings to numerics. Most of the time, you can just wire from source to destination without much worry since the functions adapt to the kind of data that you supply. How does LabVIEW know what to do? The key is object-oriented programming, where polymorphism is but one of the novel concepts

that make this new programming technology so desirable. There are, of course, limits to polymorphism, and we'll see later how to handle those special situations.

## Numeric Types

Perhaps the most fundamental data type is the **numeric**, a scalar value that may generally contain an **integer** or a **real** (floating-point) value. LabVIEW explicitly handles all the possible integer and real representations that are available on current 32-bit processors. LabVIEW 8 adds support for 64-bit integers. Figure 4.1 displays the terminals for each representation, along with the meaning of each and the number of bytes of memory occupied. LabVIEW floating-point types follow the IEEE-754 standard, which has thankfully been adopted by all major CPU and compiler designers.

The keys to choosing an appropriate numeric representation in most situations are the required **range** and **precision**. In general, the more bytes of memory occupied by a data type, the greater the possible range of values. This factor is most important with integers; among floating-point types, even single precision can handle values up to  $10^{38}$ , and it's not too often that you need such a large number. An **unsigned integer** has an upper range of  $2^N - 1$ , where  $N$  is the number of bits. Therefore, an unsigned byte ranges up to 255, an unsigned integer (2 bytes) up to 65,535, and an unsigned long integer up to 4,294,967,295. **Signed integers** range up to  $2^N - 1$ , or about one-half of their unsigned counterparts.

If you attempt to enter a value that is too large into an integer control, it will be coerced to the maximum value. But if an integer mathematical process is going on, **overflow** or **underflow** can produce erroneous results. For instance, if you do an unsigned byte addition of  $255 + 1$ , you get 0, not 256—that's overflow.

EXT	Extended-precision float (platform-dependent)	I64	Quad integer (8)
DBL	Double-precision float (8)	I32	Long integer (4)
SGL	Single-precision float (4)	I16	Integer (2)
		I8	Byte (1)
CXT	Complex extended-precision float (platform-dependent)	U64	Unsigned quad integer (8)
CDB	Complex double-precision float (16)	U32	Unsigned long integer (4)
CSG	Complex single-precision float (8)	U16	Unsigned integer (2)
		U8	Unsigned byte (1)

Figure 4.1 Scalar numerics in LabVIEW cover the gamut of integer and real types. The number of bytes of memory required for each type is shown.

Floating-point precision can be a very confusing subject, especially if you're prone to being anal-retentive. (The info-labview mailgroup has seen some lively discussions on this subject—to the point of "flame wars.") While integers by definition have a precision of exactly 1, floating-point types are *inexact* representations. The reason is that while you may be most comfortable entering and displaying a floating-point value in decimal (base-10) notation, the computer stores the value as a binary (base-2) value *with a limited number of bits*. Some rounding must occur, so this base-converted storage is therefore inexact.

For example, enter 3.3 into a single-precision, floating-point control, pop up on the control and set the precision to a large number of digits, and note that the displayed value is 3.2999999523. . . , which is just a hair different from what you entered.

The question is, How important is that rounding error to your application? You should evaluate precision in terms of **significant figures**. For instance, the number 123.450 has six significant figures. According to the IEEE-754 standard, a single-precision float guarantees 7.22 significant figures, a double-precision float gives you 15.95, and an extended-precision float gives you 19.26. When you configure a numeric indicator to display an arbitrary number of digits, all digits beyond the IEEE-specified number of significant figures are insignificant! That is, you must ignore them for the purposes of computation or display. Double-precision values are recommended for most analyses and mathematical operations to reduce cumulative rounding errors. Single-precision values are sufficient for transporting data from most real-world instruments (with simple math) since they give you a resolution of 0.1 part per million, which is better than most real, physical measurements.

IEEE floating-point numbers have a bonus feature: They can tell you when the value is invalid. If you divide by zero, the value is infinity, and a LabVIEW numeric indicator will say **inf**. Another surprisingly useful value is **NaN**, or not-a-number. It makes a great flag for invalid data.

For instance, if you're decoding numbers from a string (see the next section) and a valid number can't be found, you can have the function return NaN instead of zero or some other number that might be mistaken for real data. Also, when you are graphing data, NaN does not plot at all.

That's a handy way to edit out undesired values or to leave a blank space on a graph. You can test for NaN with the comparison function **Not A Number/Path/Refnum**, .

## Strings

### CLAD

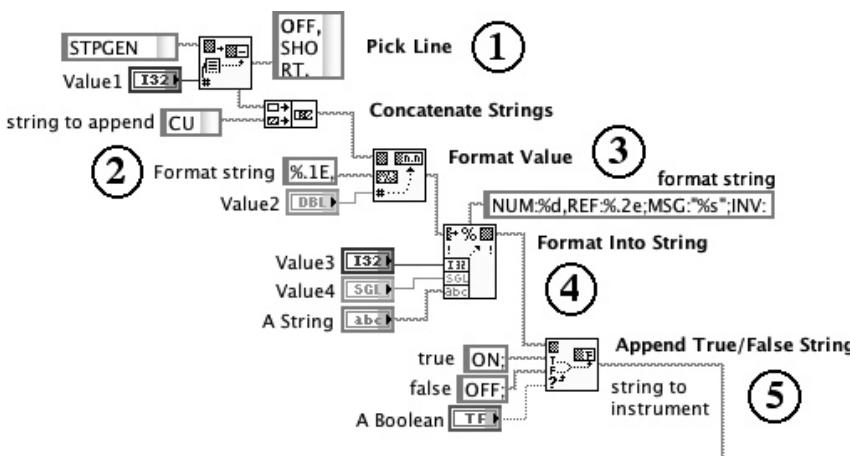
Every programmer spends a lot of time putting together strings of characters and taking them apart. Strings are useful for indicators where you need to say something to the operator, for communications

with a general-purpose interface bus (GPIB) and serial instruments, and for reading and writing data files that other applications will access. If you've had the displeasure of using *old* FORTRAN (say, the 1966 vintage), you know what it's like to deal with strings in a language that didn't even have a string or character *type*, let alone string functions. We LabVIEW users are in much better shape, with a nice set of string-wrangling functions built right in. If you know anything about the C language, some of these functions will be familiar. Let's look at some common string problems and their solutions.

## Building strings

Instrument drivers are the classic case study for string building. The problem is to assemble a command for an instrument (usually GPIB) based on several control settings. Figure 4.2 shows how string building in a typical driver works.

1. The **Pick Line** function is driven by a ring control that supplies a number (0, 1, or 2) that is mapped into the words OFF, SHORT, or LONG. These keywords are appended to the initial command string STPGEN PUL:.
2. The **Concatenate Strings** function tacks on a substring CUR:. If you need to concatenate several strings, you can resize Concatenate Strings (by dragging at any corner) to obtain more inputs.
3. The **Format Value** function is versatile for appending a single numeric value. You need to learn about C-style formatting commands



**Figure 4.2** String building in a driver VI. This one uses most of the major string-building functions in a classic diagonal layout common to many drivers.

(see the LabVIEW online help for details) to use this function. The percent sign tells it that the next few characters are a formatting instruction. What is nice about this function is that not only does it format the number in a predictable way, but also, it allows you to tack on other characters before or after the value. This saves space and gets rid of a lot of Concatenate Strings functions. In this example, the format string %e translates a floating-point value in exponential notation and adds a trailing comma.

4. A more powerful function is available to format multiple values: **Format Into String**. You can pop up on this function and select Edit Format String to obtain an interactive dialog box where you build the otherwise cryptic C-style formatting commands. In this case, it's shown formatting an integer, a float, and a string into one concatenated output with a variety of interspersed characters. Note that you can also use a control or other string to determine the format. In that case, there could be run-time errors in the format string, so the function includes error I/O clusters. It is very handy and very compact, similar to its C counterpart sprintf().
5. The **Append True/False String** function uses a boolean control to pick one of two choices, such as ON or OFF, which is then appended to the string. This string-building process may continue as needed to build an elaborate instrument command.

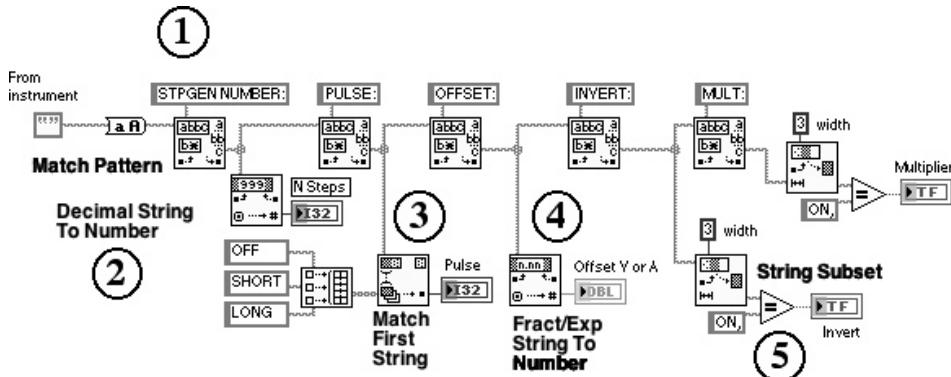
## Parsing strings

The other half of the instrument driver world involves interpreting response messages. The message may contain all sorts of headers, delimiters, flags, and who knows what, plus a few numbers or important letters that you actually want. Breaking down such a string is known as **parsing**. It's a classic exercise in computer science and linguistics as well. Remember how challenging it was to study our own language back in fifth grade, parsing sentences into nouns, verbs, and all that? We thought that was tough; then we tackled the reply messages that some instrument manufacturers come up with! Figure 4.3 comes from one of the easier instruments, the Tektronix 370A driver. A typical response message would look like this:

```
STPGEN NUMBER:18;PULSE:SHORT;OFFSET:-1.37;NVERT:OFF;MULT:ON;
```

Let's examine the diagram that parses this message.

1. The top row is based on the versatile **Match Pattern** function. Nothing fancy is being done with it here, other than searching for a desired keyword. The manual fully describes the functions of the



**Figure 4.3** Using string functions to parse the response message from a GPIB instrument. This is typical of many instruments you may actually encounter.

many special characters you can type into the *regular expression* input that controls Match Pattern. One other thing we did is to pass the incoming string through the **To Upper Case** function. Otherwise, the pattern keys would have to contain both upper- and lower-case letters. The output from each pattern match is known as *after substring*. It contains the remainder of the original string immediately following the pattern, assuming that the pattern was found. If there is any chance that your pattern might not be found, test the *offset past match* output; if it's less than zero, there was no match, and you can handle things at that point with a Case structure.

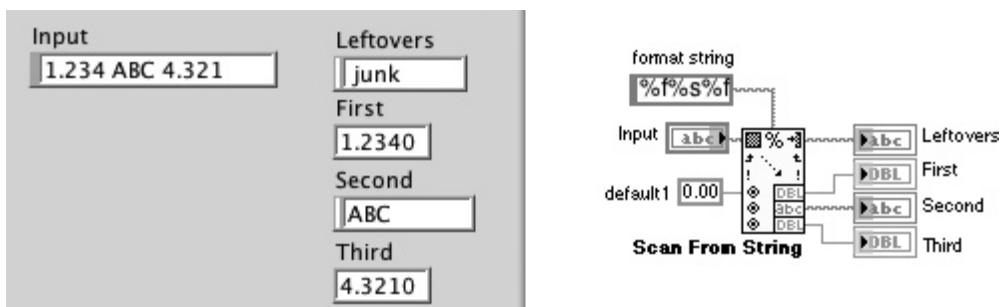
2. Each of the *after substrings* in this example is then passed to another level of parsing. The first one uses the **Decimal String To Number** function, one of several specialized number extractors; others handle octal, hexadecimal, and fraction/scientific notation. These are very robust functions. If the incoming string starts with a valid numeric character, the expected value is returned. The only problem you can run into occurs in cases where several values are run together such as [123.E3-.567.89]. Then you need to use the **Scan From String** function to break it down, provided that the format is fixed. That is, you know exactly where to split it up. You could also use Match Pattern again if there are any other known, embedded flags, even if the flag is only a space character.
3. After locating the keyword PULSE, we expect one of three possible strings: OFF, SHORT, or LONG. **Match First String** searches a string array containing these words and returns the index of the one that matches (0, 1, or 2). The index is wired to a ring indicator named **Pulse** that displays the status.

4. Keyword OFFSET is located, and the substring is passed to the **Fract/Exp String To Number** function to extract the number, which in this case is in fractional form.
5. INVERT is located next, followed by one of two possible strings, ON or OFF. When Gary first wrote this VI, he first used Index and Strip again, with its search array containing ON and OFF. But a bug cropped up! It turned out that another keyword farther along in the string (MULT) also used ON and OFF. Since Match Pattern passes us the *entire remainder* of the string, Index and Strip happily went out and found the *wrong* ON word—the one that belonged to MULT. Things were fine if the INVERT state was ON, since we found that right away. The solution he chose was to use the **String Subset** function, split off the first three characters, and test only those. You could also look for the entire command INVERT:ON or INVERT:OFF.

The moral of the story: Test thoroughly before shipping.

The **Scan From String** function, like the C `scanf()` function, can save much effort when you are parsing strings (Figure 4.4). Like its complement Format Into String, this function produces one or more polymorphic outputs (numeric or string), depending upon the contents of the *Format String* input. Input terminals allow you to determine the data format of numeric outputs and to set default values as well.

Other difficult parsing problems arise when you attempt to extract information from text files. If the person who designed the file format is kind and thoughtful, all you will have to do is to search for a keyword, then read a number. There are other situations that border on the intractable; you need to be a computer science whiz to write a reliable parser in the worst cases. These are very challenging problems, so don't feel ashamed if it takes you a long time to write a successful LabVIEW string parsing VI.



**Figure 4.4** The Scan From String function is great for parsing simple strings.

### Dealing with unprintables

Sometimes you need to create or display a string that contains some of the unprintable ASCII characters, such as control-x or the escape character. The trick is to use the pop-up item '**\ Codes Display**', which works on front panel control and indicators as well as diagram string constants. LabVIEW will interpret one- or two-character codes following the backslash character, as shown in Table 4.1. You can also enter unprintables by simply typing them into the string. Control characters, carriage returns, and so on all work fine, except for the Tab (control-i). LabVIEW uses the Tab key to switch tools, so you have to type `\t`. Note that the hexadecimal codes require uppercase letters. One other disconcerting feature of these escape codes is this: If you enter a code that can be translated into a printable character (for instance, `\41` = A), the printable character will appear as soon as you run the VI. To solve that problem, you can pop up on the string and select **Hex Display**. Then the control simply displays a series of two-digit hexadecimal values without ASCII equivalents.

There are many conversion functions in the String palette. Figure 4.5 shows just a few of them in action. These string converters are the equivalent of functions such as ASC() and CHR() in Basic, where you convert numbers to and from strings. The **Scan From String** function is particularly powerful and not limited to hexadecimal conversion. You expand it to suit the number of variables in the source string, then enter an appropriate number of format specifiers in the format string. As with its complement **Format Into String**, you can pop up on this function and select **Edit Format String** to build formatting commands.

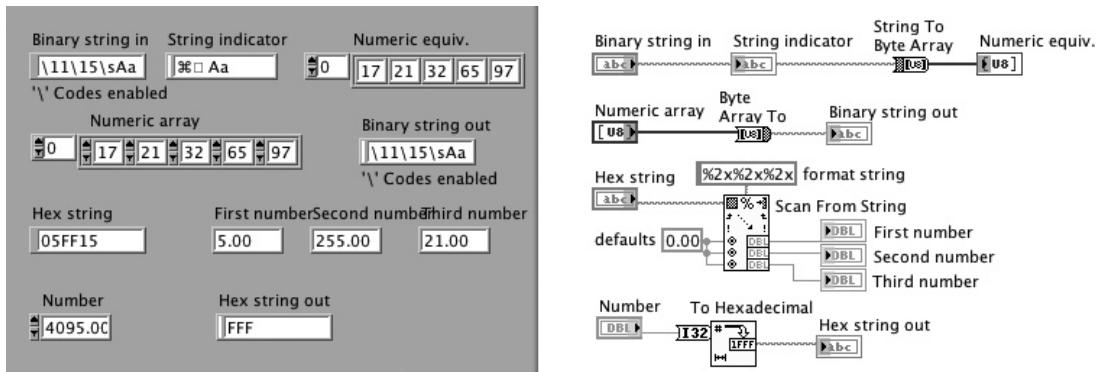
### Spreadsheets, strings, and arrays

Frequently you will need to write your data to disk for later use by spreadsheet programs and other applications that prefer tab-delimited text. Such a string might look like this:

```
value <tab> value <tab> value <cr>
```

**TABLE 4.1 Escape Sequences (\ Codes) for Strings**

Escape codes	Interpreted as
<code>\00-\FF</code>	Hexadecimal value of an 8-bit character
<code>\b</code>	Backspace (ASCII BS or equivalent to <code>\08</code> )
<code>\f</code>	Formfeed (ASCII FF or equivalent to <code>\0C</code> )
<code>\n</code>	Newline (ASCII LF or equivalent to <code>\0A</code> )
<code>\r</code>	Return (ASCII CR or equivalent to <code>\0D</code> )
<code>\t</code>	Tab (ASCII HT or equivalent to <code>\09</code> )
<code>\s</code>	Space (equivalent to <code>\20</code> )
<code>\\"</code>	Backslash (ASCII <code>\</code> or equivalent to <code>\5C</code> )



**Figure 4.5** Here are some of the ways you can convert to and from strings that contain unprintable characters. All these functions are found in the String palette.

Sometimes the delimiter is a comma or other character. No matter, LabVIEW handles all delimiters. A very important portability note is in order here. Most platforms are different with respect to their end-of-line character. A platform-independent constant, End Of Line (), is available from the String function palette that automatically generates the proper character when ported. The proper characters are

Macintosh: carriage return (\r)

PC: carriage return, then line feed (\r\n)

UNIX: line feed (\n) (ul-end)

A really interesting problem crops up with string controls when the user types a carriage return into the string: LabVIEW always inserts a *line feed*. For the examples in this book, we generally use a carriage return, since we're Mac users. But remember this portability issue if you plan to carry your VIs from one machine to another.

To convert arrays of numerics to strings, the easiest technique is to use the **Array To Spreadsheet String** function. Just wire your array into it along with a format specifier, and out pops a tab-delimited string, ready to write to a file. For a one-dimensional (1D) array (Figure 4.6), tab characters are inserted between each value, and a carriage return is inserted at the end. This looks like one horizontal row of numbers in a spreadsheet. If you don't want the default tab characters between values, you can wire a different string to the **delimiter** input.

You can read the data back in from a file (simulated here) and convert it back into an array by using the **Spreadsheet String To Array** function. It has an additional requirement that you supply a **type specifier** (such as a 1D array) to give it a hint as to what layout the data might have. The format specifier doesn't have to show the exact

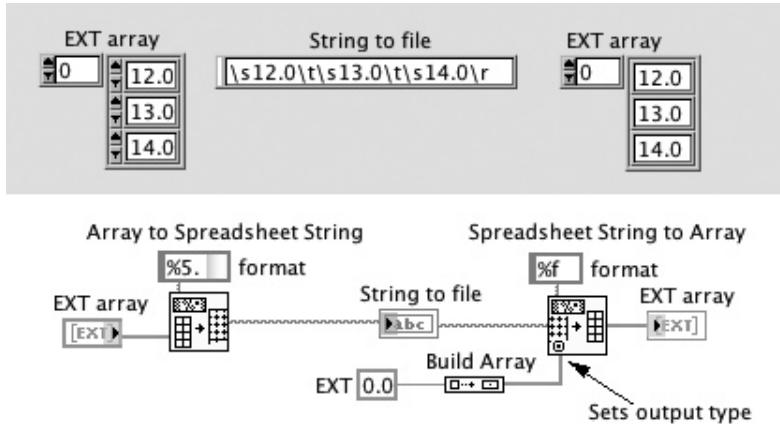


Figure 4.6 Converting a 1D array to a spreadsheet-compatible string and back.

field width and decimal precision; plain “%e” or “%d” generally does the job for floating-point or integer values, respectively.

In Figure 4.7, a two-dimensional (2D) array, also known as a **matrix**, is easily converted to and from a spreadsheet string. In this case, you get a tab between values in a row and a carriage return at the end of each row. If the rows and columns appear swapped in your spreadsheet program, insert the array function Transpose 2D Array before you convert the array to text. Notice that we hooked up Spreadsheet String To Array to a different type specifier, this time a long integer (I32). Resolution was lost because integers don’t have a fractional part; this demonstrates that you need to be careful when mixing data types. We also obtained that type specifier by a different method—an **array**

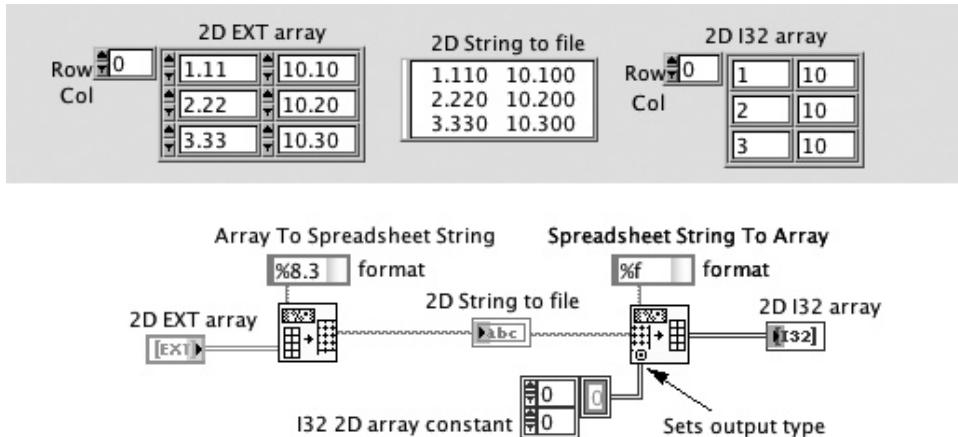
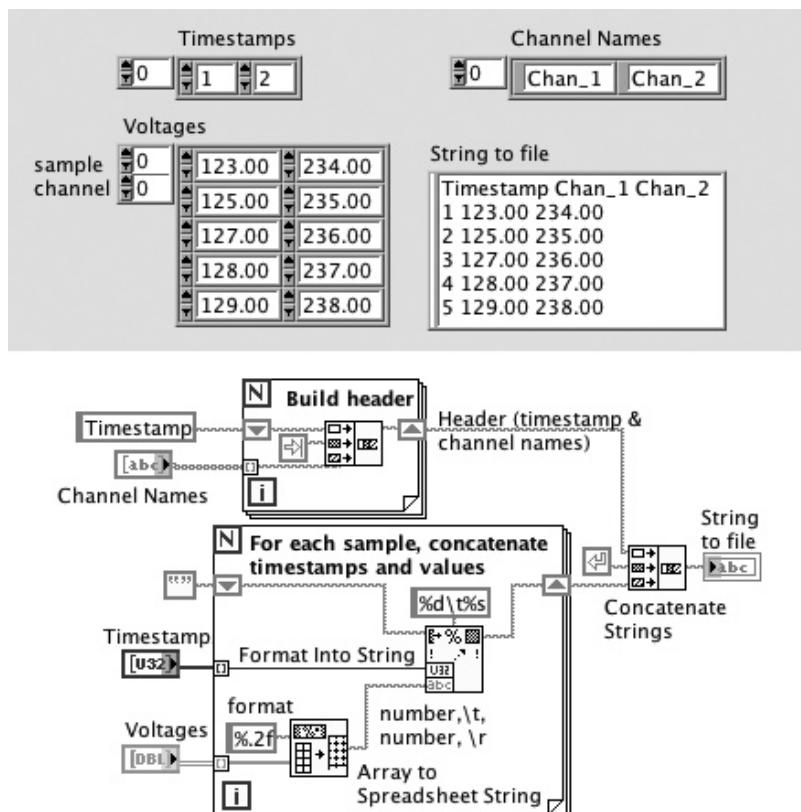


Figure 4.7 Converting an array to a table and back, this time using a 2D array (matrix).

**constant**, which you can find in the array function palette. You can create constants of *any* data type on the diagram. Look through the function palettes, and you'll see constants everywhere. In this case, choose an array constant, and then drag a numeric constant into it. This produces a 1D numeric array. Then pop up on the array and select Add Dimension to make it 2D.

A more general solution to converting arrays to strings is to use a For Loop with a shift register containing one of the string conversion functions and Concatenate Strings. Sometimes this is needed for more complicated situations where you need to intermingle data from several arrays, you need many columns, or you need other information within rows. Figure 4.8 uses these techniques in a situation where you have a 2D data array (several channels and many samples per channel), another array with time stamps, and a string array with channel names.



**Figure 4.8** A realistic example of building a spreadsheet string from a 2D data array, a timestamp array, and a string array containing channel names for the file's header. If this solution is still too slow for your application, you have to implement it in C code through a Code Interface node.

The names are used to build a header in the upper For Loop, which is then concatenated to a large string that contains the data. This business can be a real memory burner (and *slow* as well) if your arrays are large. The strings compound the speed and memory efficiency problem.

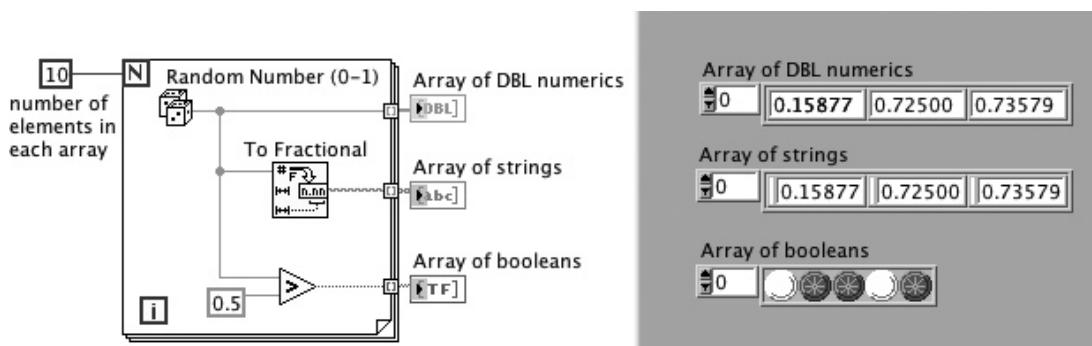
This is one of the few cases where it can be worth writing a Code Interface node, which may be the only way to obtain acceptable performance with large arrays and complex formatting requirements. If you're doing a data acquisition system, try writing out the data as you collect it—processing and writing out one row at a time—rather than saving it all until the end. On the other hand, file I/O operations are pretty slow, too, so some optimization is in order.

## Arrays

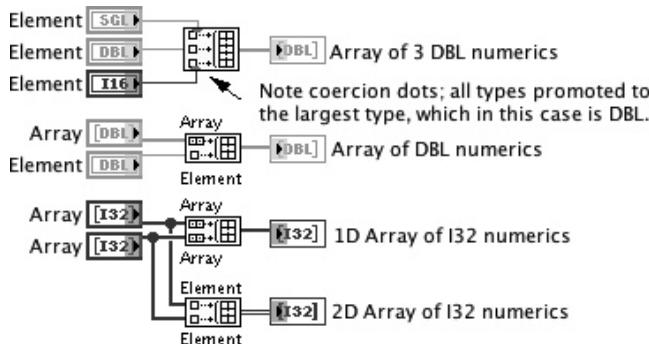
### CLAD

Any time you have a series of numbers or any other data type that needs to be handled as a unit, it probably belongs in an **array**. Most arrays are one-dimensional (1D, a column or vector), a few are 2D (a matrix), and some specialized data sets require 3D or greater. LabVIEW permits you to create arrays of numerics, strings, clusters, and any other data type (except for arrays of arrays). Arrays are often created by loops, as shown in Figure 4.9. For Loops are the best because they preallocate the required memory when they start. While Loops can't; LabVIEW has no way of knowing how many times a While Loop will cycle, so the memory manager will have to be called occasionally, slowing execution somewhat.

You can also create an array by using the **Build Array** function (Figure 4.10). Notice the versatility of Build Array: It lets you concatenate entire arrays to other arrays, or just tack on single elements. It's smart enough to know that if one input is an array and the other is a scalar, then you must be concatenating the scalar onto the array



**Figure 4.9** Creating arrays using a For Loop. This is an efficient way to build arrays with many elements. A While Loop would do the same thing, but without preallocating memory.



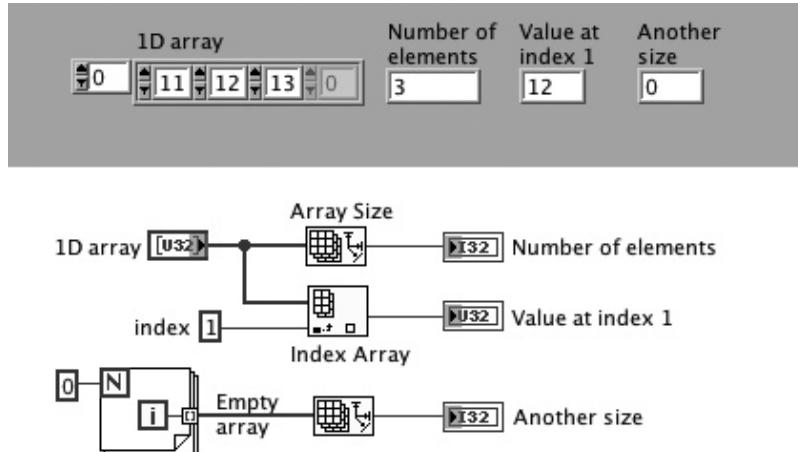
**Figure 4.10** Using the Build Array function. Use the pop-up menu on its input terminals to determine whether the input is an element or an array. Note the various results.

and the output will be similar to the input array. If all inputs have the same dimensional size (for instance, all are 1D arrays), you can pop up on the function and select *Concatenate Inputs* to concatenate them into a longer array of the same dimension (this is the default behavior). Alternatively, you can turn off concatenation to build an array with  $n + 1$  dimensions. This is useful when you need to promote a 1D array to 2D for compatibility with a VI that required 2D data.

To handle more than one input, you can resize the function by dragging at a corner. Note the **coercion dots** where the SGL (single-precision floating point) and I16 (2-byte integer) numeric types are wired to the top Build Array function. This indicates a change of data type because an array cannot contain a mix of data types. LabVIEW must promote all types to the one with the greatest numeric range, which in this case is DBL (double-precision floating point). This also implies that you can't build an array of, say, numerics and strings. For such intermixing, you must turn to **clusters**, which we'll look at a bit later.

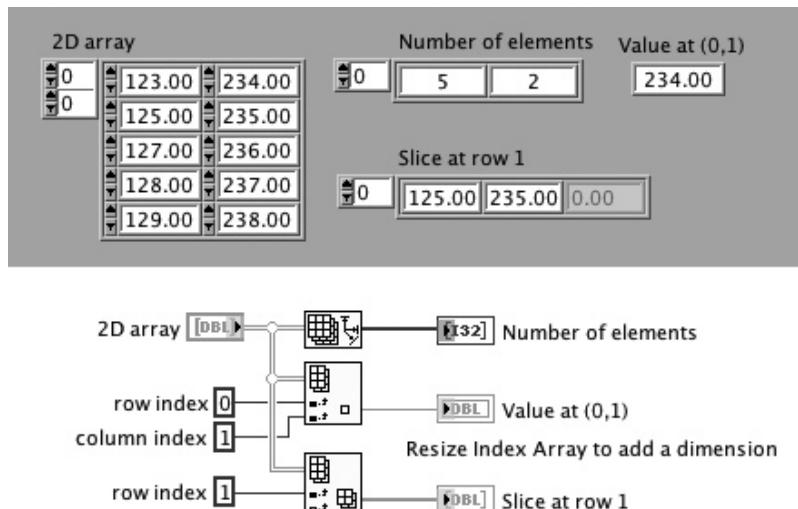
Figure 4.11 shows how to find out how many elements are in an array by using the **Array Size** function. Note that an empty array has zero elements. You can use **Index Array** to extract a single element. Like most LabVIEW functions, Index Array is polymorphic and will return a scalar of the same type as the array.

If you have a multidimensional array, these same functions still work, but you have more dimensions to keep track of (Figure 4.12). Array Size returns an array of values, one per dimension. You have to supply an indexing value for each dimension. An exception occurs when you wish to slice the array, extracting a column or row of data, as in the bottom example of Figure 4.12. In that case, you resize Array Size, but don't wire to one of the index inputs. If you're doing a lot of work with multidimensional arrays, consider creating special subVIs

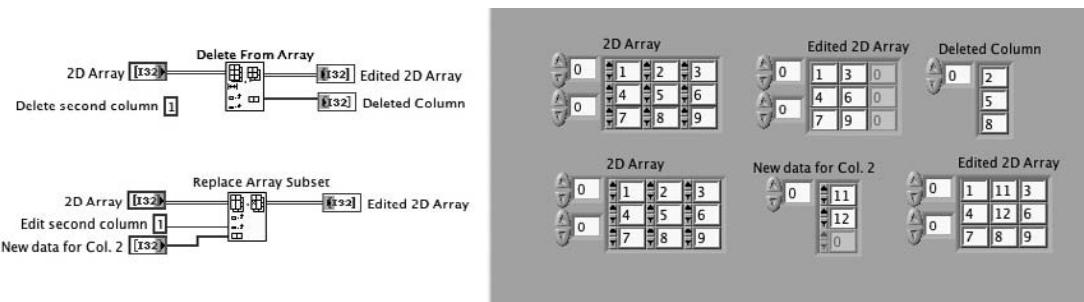


**Figure 4.11** How to get the size of an array and fetch a single value. Remember that all array indexing is based on 0, not 1.

to do the slicing, indexing, and sizing. That way, the inputs to the subVI have names associated with each dimension, so you are not forced to keep track of the purpose for each row or column. Another tip for multidimensional arrays is this: It's a good idea to place labels next to each index on an array control to remind you which is which. Labels such as *row* and *column* are useful.



**Figure 4.12** Sizing, indexing, and slicing 2D arrays are a little different, since you have two indices to manipulate at all steps.



**Figure 4.13** Two of the more powerful array editing functions are Delete From Array and Replace Array Subset. Like all the array functions, they work on arrays of any dimension.

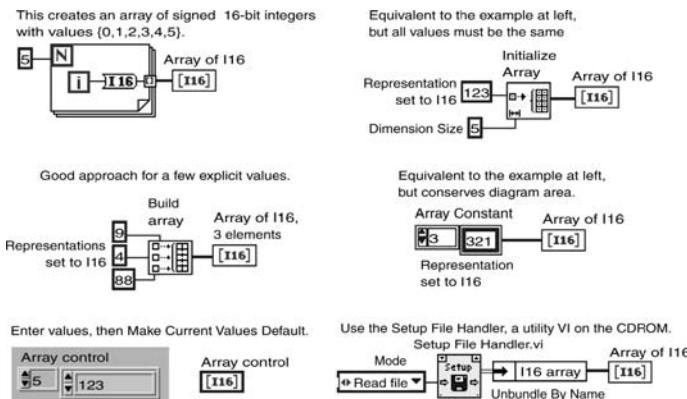
Besides slicing and dicing, you can easily do some other useful array editing tasks. In Figure 4.13, the **Delete From Array** function removes a selected column from a 2D array. The function also works on arrays of other dimensions, such as all array functions. For instance, you can delete a selected range of elements from a 1D array. In that case, you wire a value to the *Length* input to limit the quantity of elements to delete.

The **Replace Array Subset** function is very powerful for array surgery. In this example, it's shown replacing two particular values in a  $3 \times 3$  matrix. You can replace individual values or arrays of values in a single operation. One other handy function not shown here is **Insert Into Array**. It does just what you'd expect: Given an input array, it inserts an element or new subarray at a particular location. When we didn't have these functions, such actions required extensive array indexing and building, which took up valuable diagram space and added much overhead. You newcomers have it easy.

## Initializing arrays

Sometimes you need an array that is initialized when your program starts, say, for a lookup table. There are many ways to do this, as shown in Figure 4.14.

- If all the values are the same, use a For Loop with a constant inside. Disadvantage: It takes a certain amount of time to create the array.
- Use the **Initialize Array** function with the **dimension size** input connected to a constant numeric set to the number of elements. This is equivalent to the previous method, but more compact.
- Similarly, if the values can be calculated in some straightforward way, put the formula in a For Loop instead of a constant. For instance, a special waveform or function could be created in this way.



**Figure 4.14** Several ways of programmatically initializing arrays. For large amounts of data, there are tradeoffs between data storage on disk and speed of execution with each one.

- Create a diagram array constant, and manually enter the desired values. Disadvantage: It's tedious and uses memory on disk when the VI is saved.
- Create a front-panel array control and manually type in the values. Select **Make Current Value Default** from the control's Data Operations pop-up menu. From now on, that array will always have those values unless you change them. By popping up on the control or its terminal, you can select **Hide Front Panel Control** or position the control off-screen to keep anyone from modifying the data. Disadvantage: The data takes up extra space on the disk when you save the VI.
- If there is lots of data, you could save it in a file and load it at start-up.

A special case of initialization is that of an **empty array**. This is *not* an array with one or more values set to zero, false, empty string, or the like! It contains *zero* elements. In C or Pascal, this corresponds to creating a new pointer to an array. The most frequent use of an empty array is to initialize or reset a shift register that is used to hold an array. Here are some ways to create an empty array:

- Create a front-panel array control. Select **Empty Array** from its Data Operations pop-up, and then **Make Current Value Default** from the control's pop-up menu. This is, by definition, an empty array.
- Create a For Loop with the count terminal **N** wired to zero. Place a diagram constant of an appropriate type inside the loop and wire outside the loop. The loop will execute zero times (i.e., not at all), but the array that is created at the loop border tunnel will have the proper type.

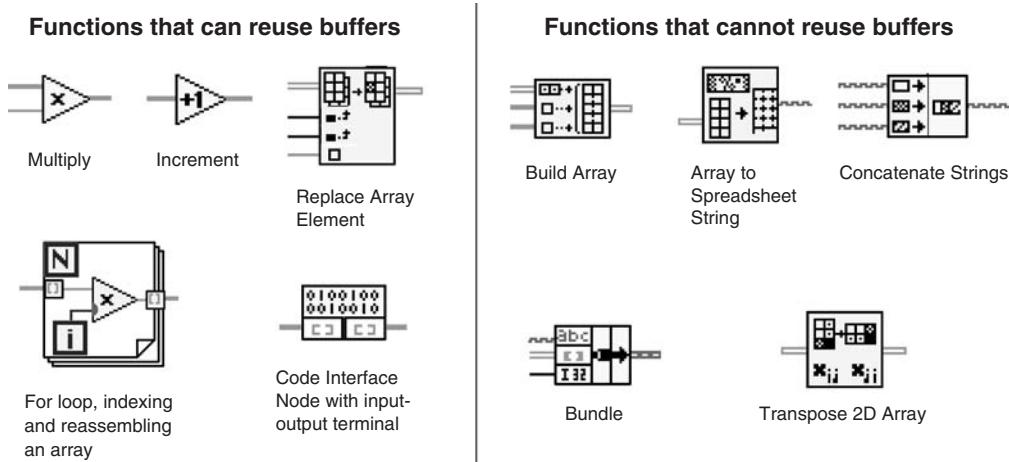
- Use the **Initialize Array** function with the **dimension size** input unconnected. This is functionally equivalent to the For Loop with  $N = 0$ .
- Use a diagram array constant. Select **Empty Array** from its Data Operations pop-up menu.

Note that you can't use the Build Array function. Its output always contains at least one element.

### Array memory usage and performance

Perhaps more than any other structure in LabVIEW, arrays are responsible for a great deal of memory usage. It's not unusual to collect thousands or even millions of data points from a physics experiment and then try to analyze or display them all at once. Ultimately, you may see a little bulldozer cursor and/or a cheerful dialog box informing you that LabVIEW has run out of memory. There are some things you can do to prevent this occurrence.

First, read the LabVIEW Technical Note *Managing Large Data Sets in LabVIEW*; we'll summarize it here, but you can find the full article online at [www.ni.com](http://www.ni.com). LabVIEW does its best to conserve memory. When an array is created, LabVIEW has to allocate a contiguous area of memory called a **data buffer** in which to store the array. (By the way, every data type is subject to the same rules for memory allocation; arrays and strings, in particular, just take up more space.) Figure 4.15 contains a sampling of functions that do and do not reuse memory in



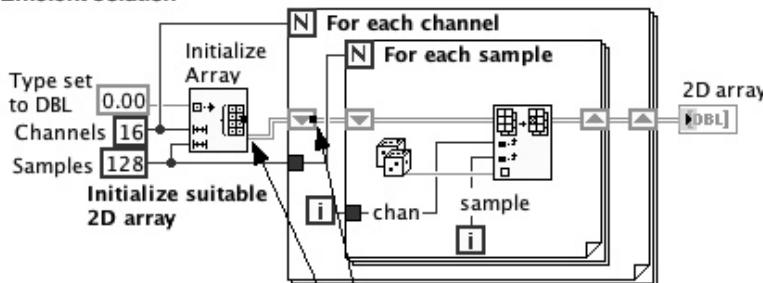
**Figure 4.15** Here are some of the operations that you can count on for predictable reuse or non-reuse of memory. If you deal with large arrays or strings, think about these differences.

a predictable way. If you do a simple operation such as multiplying a scalar by an array, no extra memory management is required. An array that is indexed on the boundary of a For Loop, processed, and then rebuilt also requires no memory management. The Build Array function, on the other hand, always creates a new data buffer. It's better to use **Replace Array Element** on an existing array, as shown in Figure 4.16. A quick benchmark of this example showed a performance increase of 30 times. This is one of the few cases where you can explicitly control the allocation of memory in LabVIEW, and it's highly advisable when you handle large arrays or when you require maximum performance.

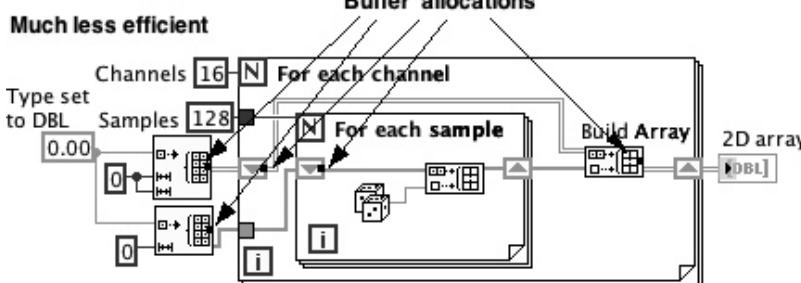
One confusing issue about multidimensional arrays involves keeping track of the indices. Which one is the row and which one is the column?

Which one does a nested loop structure act upon first? You can keep track of this by looking at the indicator on the panel, such as the one in Figure 4.16. The top index, called *channel*, is also the top index on the Replace Array Element function; this is also true of the Index Array function.

#### Efficient solution



#### Buffer allocations



**Figure 4.16** Use Replace Array Element inside a For Loop instead of Build Array to permit reuse of an existing data buffer. This is much faster. The bottom example uses five memory buffers compared to two on the top.

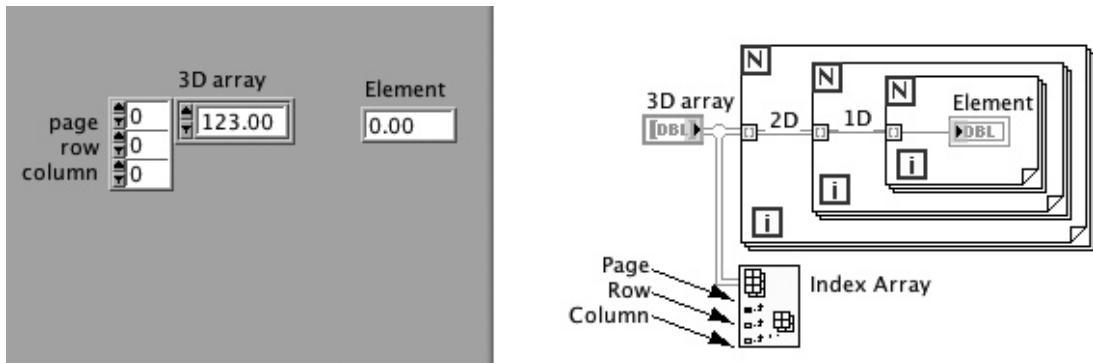


Figure 4.17 Organization of array indices.

So at least you can keep track of *that* much. By the way, it is considered good form to label your array controls and wires on the diagram as we did in this example. When you access a multidimensional array with nested loops as in the previous example, the outer loop accesses the top index and the inner loop accesses the bottom index. Figure 4.17 summarizes this array index information. The Index Array function even has pop-up tip strips that appear when you hold the wiring tool over one of the index inputs. They call the index inputs *column*, *row*, and *page*, just as we did in this example.

All this memory reuse business also adds overhead at execution time because the **memory manager** has to be called. Talk about an over-worked manager! The poor guy has to go searching around in RAM, looking for whatever size chunk the program happens to need. If a space can't be found directly, the manager has to shuffle other blocks around until a suitable hole opens up. This can take time, especially when memory is getting tight. This is also the reason your VIs sometimes execute faster the *second* time you run them: Most of the allocation phase of memory management is done on the first iteration or run.

Similarly, when an array is created in a For Loop, LabVIEW can usually predict how much space is needed and can call the memory manager just once. This is not so in a While Loop, since there is no way to know in advance how many times you're going to loop. It's also not so when you are building arrays or concatenating strings inside a loop—two more situations to avoid when performance is paramount. A new feature in LabVIEW 7.1 Professional Version shows buffer allocations on the block diagram. Select **Tools >> Advanced >> Show Buffer Allocations** to bring up the **Show Buffer Allocations** window. Black squares will appear on the block diagram, showing memory allocations. The effect is subtle, like small flakes of black pepper; but if you turn them on and off,

the allocations will jump out at you. Figure 4.16 shows what a powerful tool it can be. The best source of information on memory management is Application Note 168, *LabVIEW Performance and Memory Management*, found online at [www.ni.com](http://www.ni.com).

## Clusters

### CLAD

You can gather several different data types into a single, more manageable unit, called a **cluster**. It is conceptually the same as a *record* in Pascal or a *struct* in C. Clusters are normally used to group related data elements that are used in multiple places on a diagram. This reduces wiring clutter—many items are carried along in a single wire. Clusters also reduce the number of terminals required on a subVI. When saved as a custom control with the **typedef** or **strict typedef** option (use the **Customize Control** feature, formerly known as the **Control Editor**, to create **typedefs**), clusters serve as data type definitions, which can simplify large LabVIEW applications. Saving your cluster as a **typedef** propagates any changes in the data structure to any code using the **typedef** cluster. Using clusters is good programming practice, but it does require a little insight as to when and where clusters are best employed. If you're a novice programmer, look at the LabVIEW examples and the figures in this book to see how clusters are used in real life.

An important fact about a cluster is that it can contain only controls or indicators, but not a mixture of both. This precludes the use of a cluster to group a set of controls and indicators on a panel. Use graphical elements from the Decorations palette to group controls and indicators. If you really need to read *and* write values in a cluster, local variables can certainly do the job. We would not recommend using local variables to continuously read and write a cluster because the chance for a race condition is very high. It's much safer to use a local variable to initialize the cluster (just once), or perhaps to correct an errant input or reflect a change of mode. *Rule: For highly interactive panels, don't use a cluster as an input and output element.*

Clusters are assembled on the diagram by using either the **Bundle** function (Figure 4.18) or the **Bundle By Name** function (Figure 4.19). The data types that you connect to these functions must match the data types in the destination cluster (numeric types are polymorphic; for instance, you can safely connect an integer type to a floating-point type). The **Bundle** function has one further restriction: The elements must be connected in the proper order. There is a pop-up menu available on the cluster border called **Reorder Controls in Cluster...** that you use to set the ordering of elements. You must carefully watch cluster ordering. Two otherwise identical clusters with different element orderings can't

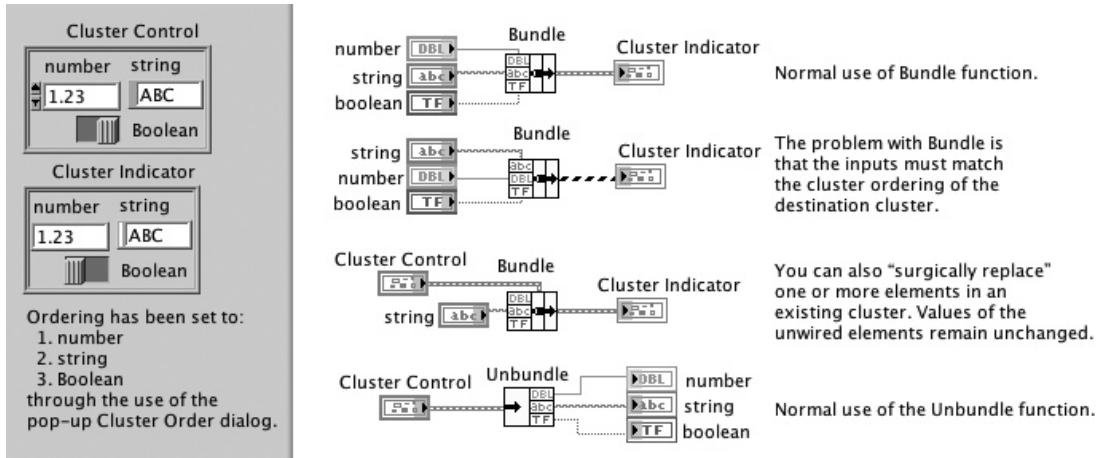


Figure 4.18 Using the Bundle and Unbundle functions on clusters.

be connected. An exception to this rule, one which causes bugs that are difficult to trace, occurs when the misordered elements are of similar data type (for instance, all are numeric). You can legally connect the misordered clusters, but element A of one cluster may actually be passed to element B of the other. This blunder is far too common and is one of the reasons for using **Bundle By Name**.

To disassemble a cluster, you can use the **Unbundle** or the **Unbundle By Name** function. When you create a cluster control, give each element a reasonably short name. Then, when you use **Bundle By Name** or **Unbundle By Name**, the name doesn't take up too much space on the diagram. There is a pop-up menu on each of these functions (**Select Item**) with which you select the items to access. Named access has the

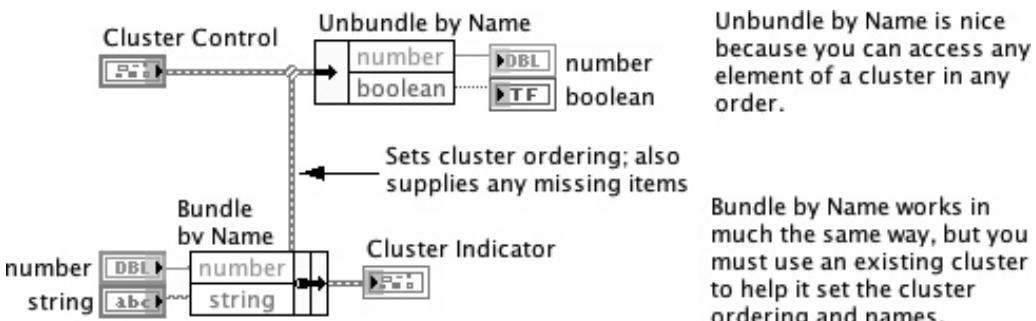


Figure 4.19 Use Bundle By Name and Unbundle By Name in preference to their no-name counterparts.

additional advantage that adding an item to the related cluster control or indicator doesn't break any wires as it does with the unnamed method. *Rule: Always use named cluster access except when there is some compelling reason not to.*

When you use Bundle By Name, its middle terminal must be wired. The functions of the middle terminal on Bundle By Name are (1) to determine the element ordering and the data types and (2) to set the item names. Even if you have wired all input elements, you must wire the middle terminal because the input elements determine only the data types. The Bundle function does not have this limitation; you need only wire to its middle terminal when you wish to access a limited set of a cluster's elements.

Clusters are often incorporated into arrays (we call them cluster arrays), as shown in Figure 4.20. It's a convenient way to package large collections of data such as I/O configurations where you have many different pieces of data to describe a channel in a cluster and many channels (clusters) in an array. Cluster arrays are also used to define many of the graph and chart types in LabVIEW, a subject discussed in detail in Chapter 20, "Data Visualization, Imaging, and Sound." Figure 4.20 also shows the LabVIEW equivalent of an *array of arrays*, which is implemented as an array of clusters of arrays. Note the difference between this construct and a simple 2D array, or matrix. Use these arrays of cluster arrays when you want to combine arrays with different sizes; multidimensional LabVIEW arrays are always rectangular.

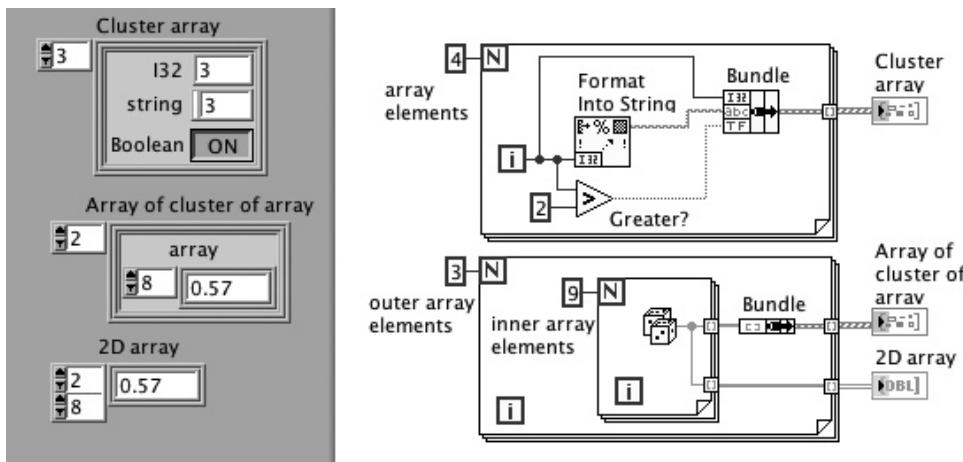


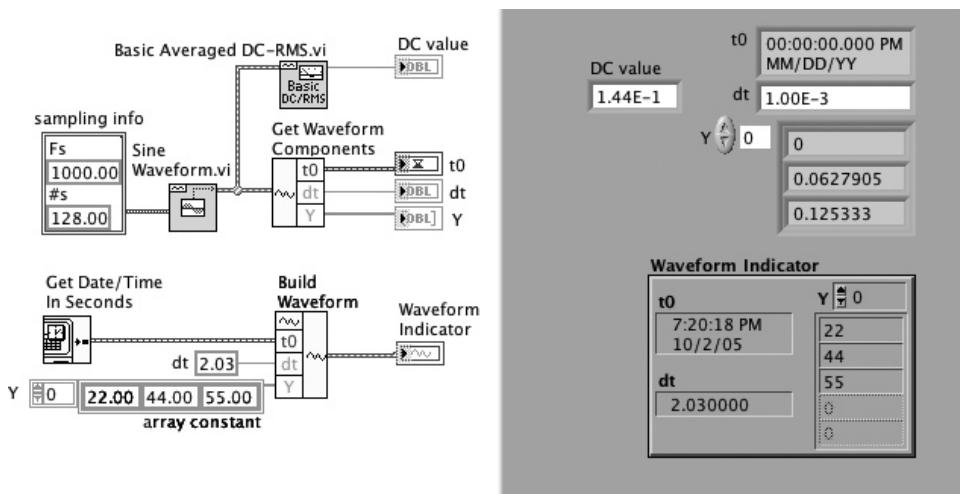
Figure 4.20 Building a cluster array (top) and building an array of clusters that contain an array (bottom). This is distinctly different from a 2D array.

## Waveforms

### CLAD

A handy data type, introduced in LabVIEW 6, is the **Waveform**. It's a sensible grouping of information that describes the very common situation of a one-dimensional time-varying waveform. The waveform type is similar to a cluster containing a 1D numeric array (the data), timing information (start time and sample interval), and a variant part containing user-definable items such as the channel's name, units, and error information. Native waveform controls and indicators are available from the I/O control palette, but other indicators, such as graphs, will also adapt directly to waveforms. In the Programming palette, there's a subpalette called Waveform that includes a variety of basic operations, such as building, scaling, mathematical operations, and so forth, plus higher-level functions, such as waveform generation, measurements, and file I/O.

Figure 4.21 shows some of the basic things you might do with a waveform. Waveforms can come from many sources, among them the **Waveform Generation** VIs, such as the **Sine Waveform VI** that appears in the figure. The analog data acquisition (DAQ) VIs also handle waveforms directly. Given a waveform, you can either process it as a unit or disassemble it for other purposes. In this example, we see one of the **Waveform Measurement** utilities, the **Basic Averaged DC-RMS VI**, which extracts the dc and rms ac values from a signal. You can explicitly access waveform components with the **Get Waveform Components** function, which looks a lot like Unbundle By Name.



**Figure 4.21** Here are the basics of the waveform data type. You can access individual components or use the waveform utility VIs to generate, analyze, or manipulate waveforms.

To assemble a waveform data type from individual components, use the **Build Waveform** function. Not all inputs need be supplied; quite often, the starting time ( $t_0$ ) can be left unwired, in which case it will be set to zero.

Thanks to polymorphism, waveforms can be wired to many ordinary functions, but there are some limits and special behavior, as shown in Figure 4.22. To add a constant offset to each data value in a waveform, you can use the regular Add node. Another common operation is to sum the data for two waveforms (for instance, to add uniform noise to a synthetic waveform). In this case, the Add node doesn't do what you expect. The result of adding a waveform and a numeric array is counterintuitive: You end up with an array of waveforms with the same number of waveforms as there are elements in the numeric array. Instead, you must use the Build Waveform function to promote your numeric array to the waveform type, and then you use the Add node. Another method is to unbundle the Y array by using Get Waveform Components, add the two arrays with the Add function, and reassemble with Build Waveform. LabVIEW 8 correctly handles the situation where you use the Add node to directly add two waveforms. Both waveforms must have the same dt values. The resulting waveform retains the t0 and **attributes** of the waveform connected to the upper terminal of the Add node.

So what about these waveform attributes? That component is primarily for use with the DAQ functions when you have defined channel names and units through the DAQ configuration utilities. In that case, the channel information is passed along with acquired analog data. But you can also insert your own information into this **variant** data field, and read it back as well. Variant data types were originally added to LabVIEW to handle the complicated data that is sometimes required by **ActiveX** objects. Like a cluster, a variant is an arbitrary collection of items, but a variant lets you change the data type (add or delete items) while a VI is running.

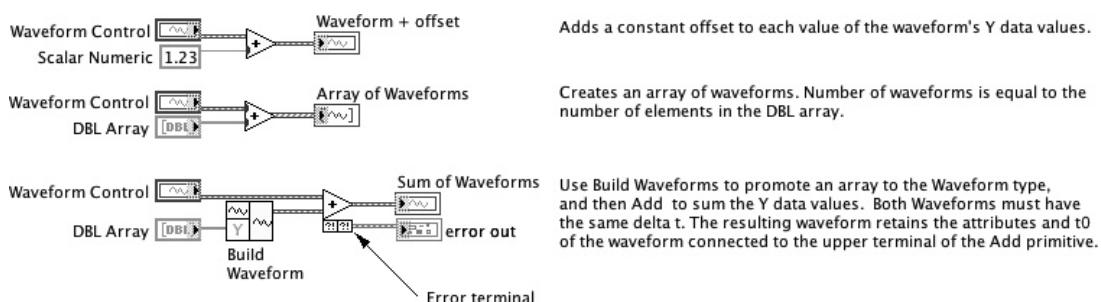
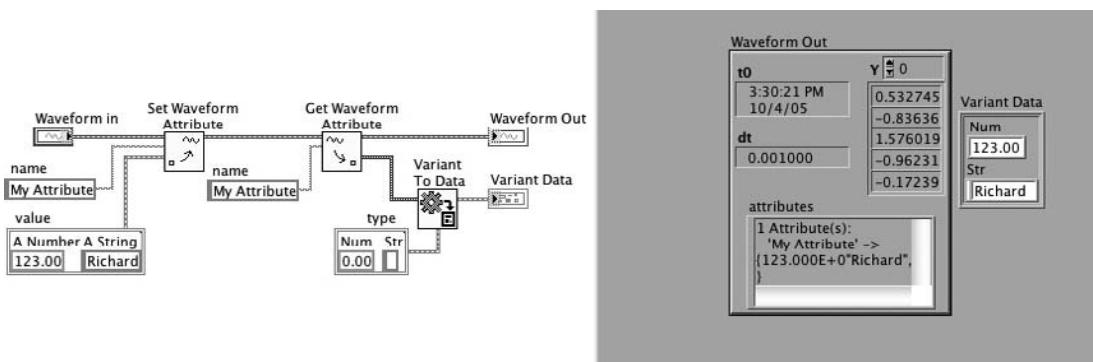


Figure 4.22 Polymorphism permits standard operations on waveforms.

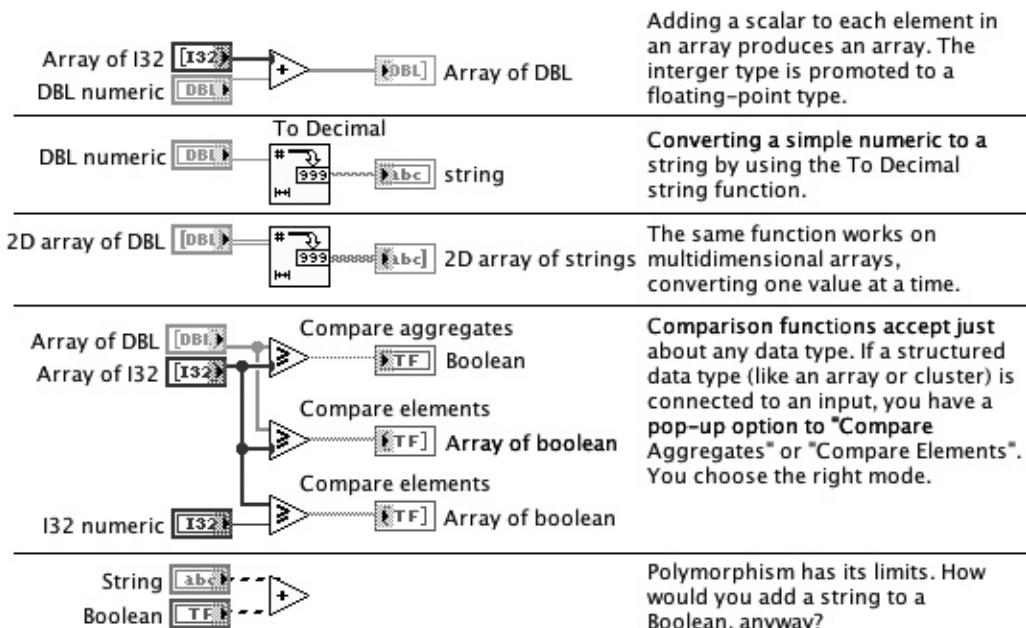


**Figure 4.23** The stealth component of waveforms, called *attributes*, can be accessed when you need them. They are *variant* data types, which require some care during access.

In Figure 4.23, we use **Set Waveform Attribute** to create a new attribute called My Attribute that is a cluster with a number and a string inside. This information rides along with that waveform until it is either modified or deleted. If you use one of the waveform file I/O utilities, all attributed data will be stored in the file for you. You can also display it in a waveform indicator, as we have in this figure. Pop up on the waveform indicator, and under **Visible Items** you can choose **Attributes**. To read the attribute information back into your program, use the **Get Waveform Attribute** function. If you supply the name of an existing attribute, the function returns exactly what you asked for. If you don't supply a name, it returns all available attributes. In either case, the information comes out in the form of a variant. To deal with that, use the **Variant To Data** function from the Cluster and Variant palette. This little fellow converts arbitrary variant data to whatever type you wish. In this case, the type is set to a cluster of the right kind, and the information you put in comes back out as desired. What if you give it the wrong data type? Conversion fails, and Variant To Data returns an error cluster that tells you what happened.

## Data Type Conversions

Thanks to polymorphism, you can usually wire from source to destination without much worry since the functions adapt to the kind of data that you supply. For instance, in the top part of Figure 4.24, a constant is added to each value in an array. That's a wonderful time saver compared with having to create a loop to iterate through each element of the array. There are, of course, limits to polymorphism, as the bottom example in Figure 4.24 shows. The result of adding a boolean to a string is a little



**Figure 4.24** Polymorphism in action. Think how complicated this would be if the functions didn't adapt automatically. The bottom example shows the limits of polymorphism. How in the world could you add a boolean or a string? If there *were* a way, it would be included in LabVIEW.

hard to define, so the natural polymorphism in LabVIEW doesn't permit these operations. But what if you actually *needed* to perform such an operation? That's where conversions and type casting come in.

### Conversion and coercion

#### CLAD

Data in LabVIEW has two components, the **data** and **type descriptor**. You can't see the type descriptor; it is used internally to give LabVIEW directions on how to handle the associated data—that's how polymorphic functions know what kind of data is connected. A type descriptor identifies the type of the data (such as a DBL floating-point array) and the number of bytes in the data. When data is **converted** from one type to another, both the data component and the type descriptor are modified in some fashion. For example,  [I32] → [DBL], where an I32 (signed integer with 32 bits, or 4 bytes) is converted to a DBL (64-bit, or 8-byte, floating point), the value contained in the I32 is changed into a *mantissa* and an *exponent*. The type descriptor is changed accordingly, and this new data type takes up a bit more memory. For conversion between scalar numeric types, the process is very simple, generally requiring only one CPU instruction. But if aggregate types are involved (strings, clusters, etc.), this conversion process takes some time.

First, the value has to be interpreted in some way, requiring that a special conversion program be run. Second, the new data type may require more or less memory, so the system's memory manager may need to be called. By now you should be getting the idea that conversion is something you may want to avoid, if only for performance reasons.

Conversion is explicitly performed by using one of the functions from the Conversion menu. They are polymorphic, so you can feed them scalars (simple numbers or booleans), arrays, clusters, and so on, as long as the input makes some sense. There is another place that conversions occur, sometimes without you being aware. When you make a connection, sometimes a little gray dot appears at the destination's terminal: . This is called a coercion dot, and it performs exactly the same operation as an explicit conversion function. One other warning about conversion and coercion: *Be wary of lost precision.*

A DBL or an EXT floating point can take on values up to  $10^{-237}$  or thereabouts.

If you converted such a big number to an unsigned byte (U8), with a range of only 0 to 255, then clearly the original value could be lost. It is generally good practice to modify numeric data types to eliminate coercion because it reduces memory usage and increases speed. Use the **Representation** pop-up menu item on controls, indicators, and diagram constants to adjust the representation.

### Intricate conversions and type casting

Besides simple numeric type conversions, there are some more advanced ones that you might use in special situations. This is an advanced topic, so we've saved it for last.

Figure 4.25 uses an intertype conversion to make a cluster of booleans into an array. **Cluster To Array** works on any cluster that contains

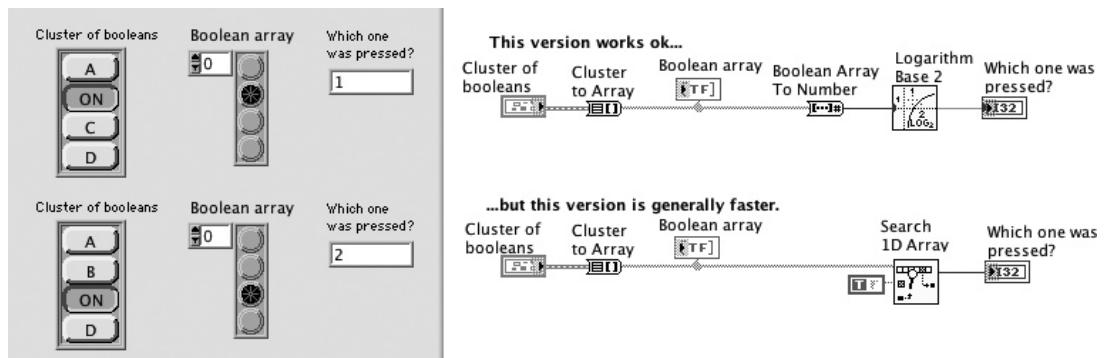


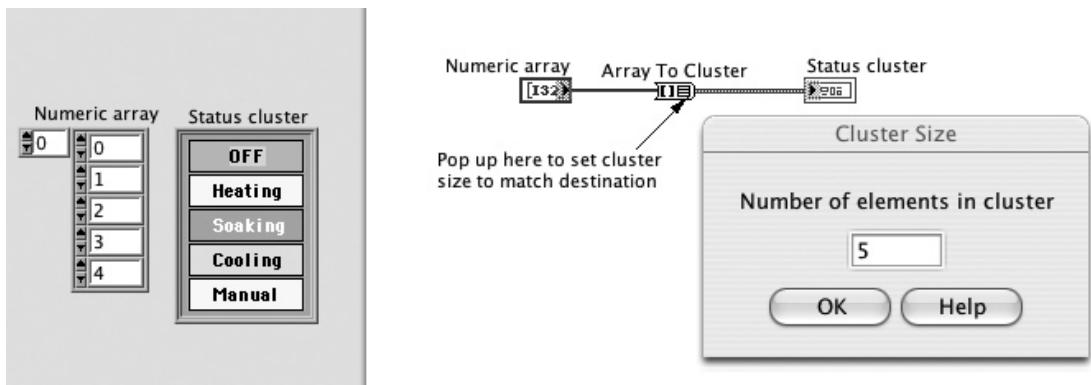
Figure 4.25 A cluster of booleans makes a nice user-interface item, but it's hard to interpret. Here are two solutions that find the true bit. The bottom solution turned out to be the faster.

only controls of the same type (you can't arbitrarily mix strings, numerics, etc.). Once you have an array of booleans, the figure shows two ways to find the element that is True. In the upper solution, the **Boolean Array To Number** function returns a value in the range of 0 to  $2^{32} - 1$  based on the bit pattern in the boolean array. Since the bit that is set must correspond to a power of 2, you then take the  $\log_2$  of that number, which returns a number between 0 and 32 for each button and -1 for no buttons pressed. Pretty crafty, eh? But the bottom solution turns out to be faster. Starting with the boolean array, use Search 1D Array to find the first element that is true. Search 1D Array returns the element number, which is again a number between 0 and 32, or a large negative number for no buttons.

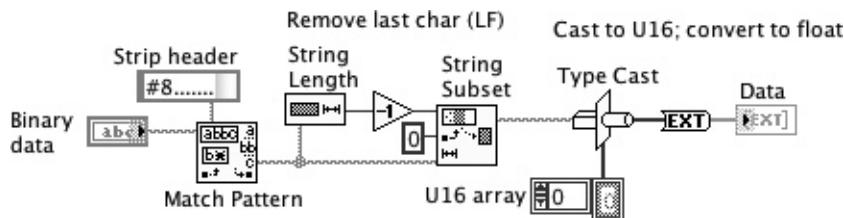
This number could then be passed to the selection terminal in a Case structure to take some action according to which switch was pressed.

Figure 4.26 shows a way to use a nice-looking set of **Ring Indicators** in a cluster as a status indicator. An array of I32 numerics is converted to a cluster by using the **Array To Cluster** function. A funny thing happens with this function. How does LabVIEW know how many elements belong in the output cluster (the array can have any number of elements)? For this reason, a pop-up item on Array To Cluster called **Cluster Size** was added. You have to set the number to match the indicator (5, in this case) or you'll get a broken wire.

One of the most powerful ways to change one data type to another is **type casting**. As opposed to conversions, type casting changes only the type descriptor. *The data component is unchanged.* The data is in no way rescaled or rearranged; it is merely interpreted in a different way. The good news is that this process is very fast, although a new copy of



**Figure 4.26** A numeric array is converted to a cluster of Ring Indicators (which are of type I32) by using Array To Cluster. Remember to use the pop-up item on the conversion function called Cluster Size for the number of cluster elements. In this case, the size is 5.



**Figure 4.27** Lifted from the HP 54510 driver, this code segment strips the header off a data string, removes a trailing linefeed character, then type casts the data to a U16 integer array, which is finally converted to an EXT array.

the incoming data has to be made, requiring a call to the memory manager. The bad news is that you have to know what you're doing! Type casting is a specialized operation that you will very rarely need, but if you do, you'll find it on the **Data Manipulation** palette. LabVIEW has enough polymorphism and conversion functions built in that you rarely need the Type Cast function.

The most common use for the Type Cast function is shown in Figure 4.27, where a binary data string returned from an oscilloscope is type cast to an array of integers. Notice that some header information and a trailing character had to be removed from the string before casting. Failure to do so would leave extra garbage values in the resultant array. Worse yet, what would happen if the incoming string were off by 1 byte at the beginning? Byte pairs, used to make up I16 integers, would then be incorrectly paired. Results would be very strange. Note that the Type Cast function will accept most data types except for clusters that contain arrays or strings.

*Warning:* The Type Cast function expects a certain byte ordering, namely **big-endian**, or most significant byte first to guarantee portability between platforms running LabVIEW. But there are problems interpreting this data *outside* of LabVIEW. Big-endian is the normal ordering for the Macintosh and Sun, but not so on the PC! This is an example where your code, or the data you save in a binary file, may not be machine-independent.

Indeed, there is much trouble in type casting land, and you should try to use polymorphism and conversion functions whenever possible. Consider Figure 4.28. The first example uses the Type Cast function, while the second example accomplishes exactly the same operation—writing a binary image of an array to disk—in a much clearer, more concise way. Keep looking through the function palettes if the particular data type compatibility that you need is not apparent. The bottom example in Figure 4.28 shows how flexible the LabVIEW primitives are. In this case, the Write File function accommodates any imaginable data type without putting you through type casting obfuscation.

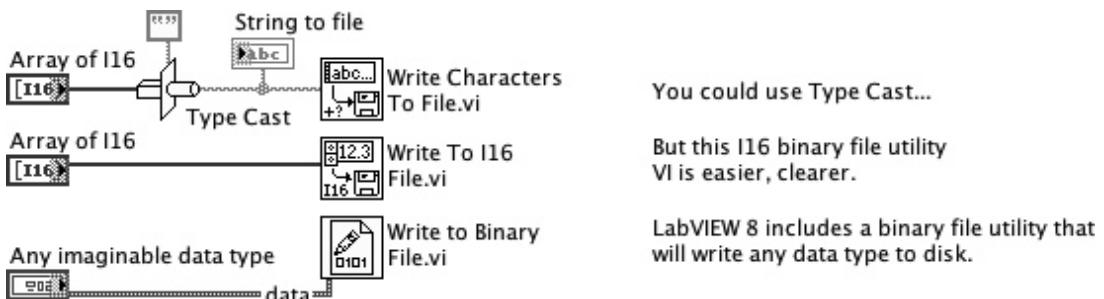


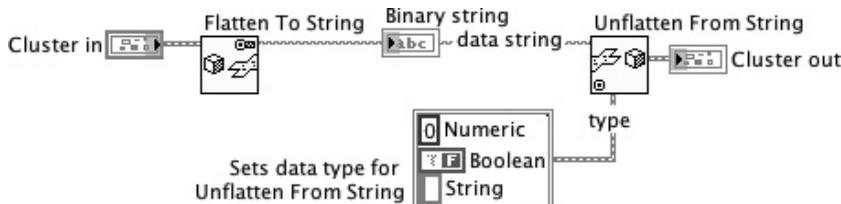
Figure 4.28 “Eschew obfuscation,” the English teacher said. Look for ways to avoid type casting and simplify your diagrams. Higher-level functions and primitives are extremely flexible.

If you get into serious data conversion and type casting exercises, be careful, be patient, and prepare to explore the other data conversion functions in the Data Manipulation palette. Sometimes the bytes are out of order, as in the PC versus Mac situation. In that case, try **Swap Bytes** or **Swap Words**, and display the data in a numeric indicator with its format set to hexadecimal, octal, or binary (use the pop-up menu **Format and Precision**). Some instruments go so far as to send the data backward; that is, the first value arrives last. You can use **Reverse Array** or **Reverse String** to cure that nasty situation. **Split Number** and **Join Number** are two other functions that allow you to directly manipulate the ordering of bytes in a machine-independent manner. Such are the adventures of writing instrument drivers, a topic covered in detail in Chapter 10, “Instrument Driver Basics.”

### Flatten To String (. . . Do what?)

We mentioned before that the Type Cast function can't handle certain complicated data types. That's because LabVIEW stores strings and arrays in *handle blocks*, which are discontiguous segments of memory organized in a tree structure. When such data types are placed in a cluster, the data may be physically stored in many areas of memory. Type Cast expects all the data to be located in a single contiguous area so that it can perform their simple and fast transformation. You can read all about LabVIEW data storage details online in Application Note AN154, *LabVIEW Data Storage*.

Occasionally you need to transmit arbitrarily complex data types over a serial link. The link may be a serial port, a network connection using a protocol such as TCP/IP, or even a binary file on disk, which is, in fact, a serial storage method. Type Cast can't do it, but the **Flatten To String** function can. What this function does is copy all discontiguous data into one contiguous buffer called the **data string**.



**Figure 4.29** Use *flattened* data when you need to transmit complicated data types over a communications link or store them in a binary file. LabVIEW 8 lets you set byte order and optionally prepend the data string's size.

The data string also contains embedded header information for nonscalar items (strings and arrays), which are useful when you are trying to reconstruct flattened data. Figure 4.29 shows **Flatten To String** in action, along with its counterpart **Unflatten From String**. The data string could be transmitted over a network, or stored in a file, and then read and reconstructed by **Unflatten From String**. As with all binary formats, you must describe the underlying data format to the reading program. **Unflatten From String** requires a data type to properly reconstruct the original data. In LabVIEW 7.x and previous versions, the data type descriptor was available as an I16 array. LabVIEW 8's data type descriptors are 32-bit, and the type descriptor terminal is no longer available on **Flatten to String**. If you still need the type descriptor, select **Convert 7.x Data** from **Flatten to String**'s pop-up menu.

The most common uses for flattened data are transmission over a network or storage to a binary file. A utility VI could use this technique to store and retrieve clusters on disk as a means of maintaining front-panel setup information.

### Enumerated types (enums)



A special case of an integer numeric is the **enum**, or enumerated type. It's a notion borrowed from C and Pascal where each consecutive value of an integer, starting with zero, is assigned a name. For instance, red, green, and blue could correspond to 0, 1, and 2. You create an enum by selecting an enum control from the **Ring and Enum** control palette, and then typing in the desired strings. The handiest use for an enum is as a selector in a Case structure because the name shows up in the header of the Case, thus making your program self-documenting (Figure 4.30). Enums can be compared with other enums, but the comparisons are actually based on the integer values behind the scenes, not the strings that you see. So you don't need to do any special conversion to compare two dissimilar enums or to compare an enum with an integer.

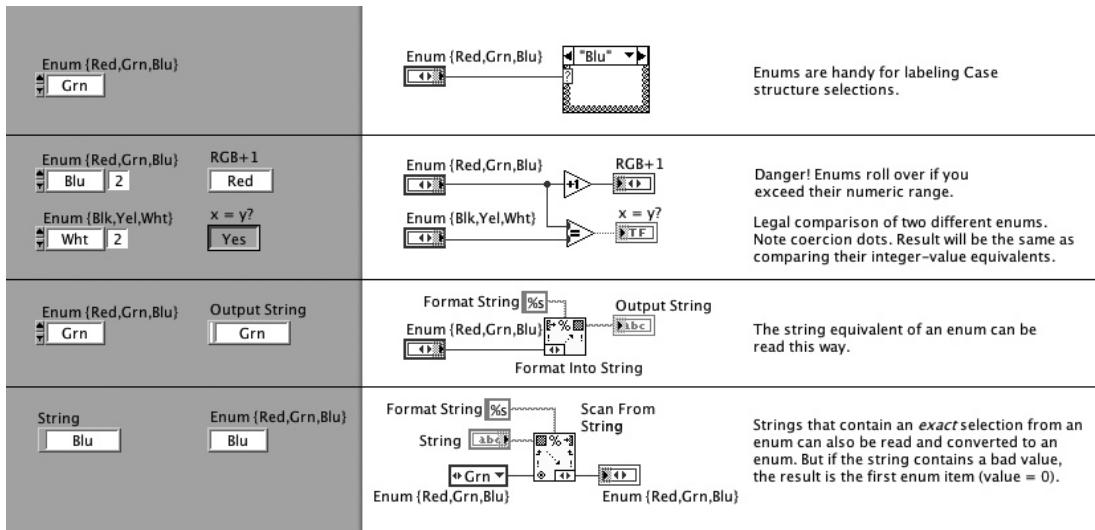


Figure 4.30 Enumerated types (enums) are integers with associated text. Here are some useful tips for comparison and conversion.

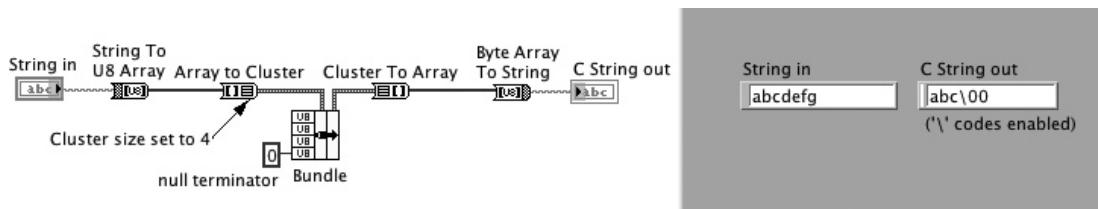
Figure 4.30 shows some precautions and tricks for enums. One surprise comes when you attempt to do math with enums. They're just integers, right? Yes, but their range is limited to the number of enumerated items. In this example, there are three values {Red, Grn, Blu}, so when we attempt to add one to the maximum value (Blu, or 2), the answer is Red—the integer has rolled over to 0.

You can also do some nonobvious conversions with enums, as shown in the bottom of Figure 4.30. Given an enum, you can extract the string value by using **Format Into String** with a format specifier of `%s`.

Similarly, you can match an incoming string value with the possible string equivalents in an enum by using **Scan From String**. There is a risk of error here, however: The incoming string has to be an exact case-sensitive match for one of the possible enum values. Otherwise Scan From String will return an error, and your output enum will be set to the value of the input enum. To distinguish this result from a valid conversion, you could add a value to your enum called *Invalid* and handle that value appropriately.

## Get Carried Away Department

Here's a grand finale for conversion and type casting. Say that you need to fix the length of a string at three characters and add on a *null* (zero) terminator. Gary actually had to do this for compatibility with



**Figure 4.31** Making a fixed-length, null-terminated string for compatibility with the C language. The key is the **Array To Cluster** conversion that produces a fixed number of elements. This is a really obscure example.

another application that expected a C-style string, which requires the addition of a null terminator. Let's use everything we know about the conversion functions and do it as in Figure 4.31.

This trick (courtesy of Rob Dye, a member of the LabVIEW development team) uses the Set Cluster Size pop-up setting in an **Array To Cluster** conversion function to force the string length to be four characters, total. The bundler allows parallel access to the whole string, so it's easy to change one character, in this case the last one. It turns out that this program is very fast because all the conversions performed here are actually type casting operations underneath. The disadvantage of this technique, as compared to using a few string functions, is that it is hard to understand.

*This page intentionally left blank*

# Timing

Software-based measurement and control systems have requirements for timing and synchronization that are distinctly different from those of ordinary applications. If you're writing an application that serves only a human operator (such as a word processor), chances are that your only timing requirement is to be fast enough that the user doesn't have to sit and wait for the computer to catch up. Exactly *when* an event occurs is not so important. But in a control system, there's a physical process that demands regular service to keep those boiling kettles from overflowing. Similarly, a data acquisition system needs to make measurements at highly regulated intervals, lest the analysis algorithms get confused. That's why LabVIEW has built-in timing functions to measure and control time. Depending on your needs, LabVIEW's timing functions can be simple and effective or totally inadequate. And the problems are not all the fault of LabVIEW itself; there are fundamental limitations on all general-purpose computer systems with regard to real-time response, whatever *that* means (1 second? 0.01 second? 1 ns?). Most applications we've seen work comfortably with the available LabVIEW time measurements that resolve milliseconds, and many more operate with 1-s resolution. A few applications demand submillisecond resolution and response time, which is problematic owing primarily to operating system issues. Those cases require special attention. So, what's with all this timing stuff, anyway?

## Where Do Little Timers Come From?

When LabVIEW was born on the Macintosh, the only timer available was based on the 60-Hz line frequency. Interrupts to the CPU occurred every 1/60 s and were counted as long as the Mac was powered on.

The system timer on modern computers is based on a crystal-controlled oscillator on the computer's motherboard. Special registers count each tick of the oscillator. The operating system abstracts the tick count for us as milliseconds and microseconds. It's funny how we instrumentation specialists benefited from the consumer-oriented multimedia timing. Without the need for high resolution time for audio and video, we may well have been stuck at 60 Hz. One might argue that all those ultrafast video accelerators and even the speed of CPUs are driven primarily by the enormous video game market!

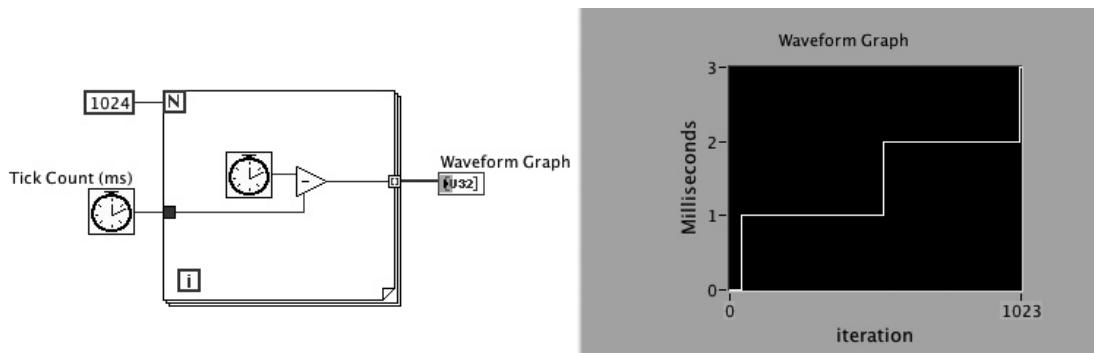
Three fine-resolution timers are available within LabVIEW on all platforms: **Wait (ms)**, **Wait Until Next ms Multiple**, and **Tick Count (ms)**. These functions attempt to resolve milliseconds within the limitations of the operating systems. Two new timing functions, **Timed Loops** and **Timed Sequence Structure**, are deterministic programming structures similar to While Loops and Sequence structures. The determinism and resolution of these two structures are operating system- and hardware platform-dependent. On a PC running Windows, the Timed Loop has a resolution of 1 ms; but under a real-time operating system (RTOS) resolution can be 1  $\mu$ s. Timed Loops also have other exciting features such as the ability to synchronize to a data acquisition task.

Another kind of timer that is available in LabVIEW is the system clock/calendar which is maintained by a battery-backed crystal oscillator and counter chip. Most computers have a clock/calendar timer of this kind. The timing functions that get the system time are **Get Date/Time In Seconds**, **Get Date/Time String**, and **Seconds To Date/Time**. These functions are as accurate as your system clock. You can verify this by observing the drift in the clock/calendar displayed on your screen. Unless you synchronize to a network time source, it's probably within a minute per month or so.

Are you curious about the timing resolution of your system? Then write a simple timer test VI like the one shown in Figure 5.1. A For Loop runs at maximum speed, calling the timing function under test. The initial time is subtracted from the current time, and each value is appended to an array for plotting. The system in Figure 5.1 has a resolution of 1 ms and took approximately 1  $\mu$ s per iteration.

## Using the Built-in Timing Functions

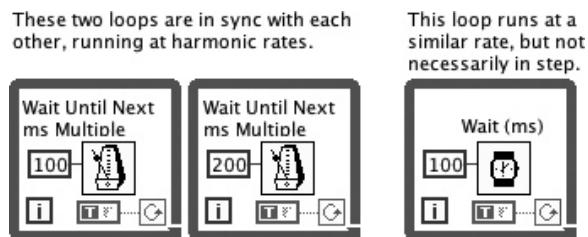
There are two things you probably want to do with timers. First, you want to make things happen at regular intervals. Second, you want to record when events occur.



**Figure 5.1** Write a simple timer test VI to check the resolution of your system's LabVIEW ticker.

## Intervals

Traditionally if you want a loop to run at a nice, regular interval, the function to use is Wait Until Next ms Multiple. Just place it inside the loop structure, and wire it to a number that's scaled in milliseconds. It waits until the tick count in milliseconds becomes an exact multiple of the value that you supply. If several VIs need to be synchronized, this function will help there as well. For instance, two independent VIs can be forced to run with harmonically related periods such as 100 ms and 200 ms, as shown in Figure 5.2. In this case, every 200 ms, you would find that both VIs are in sync. The effect is exactly like the action of a metronome and a group of musicians—it's their heartbeat. This is not the case if you use the simpler Wait (ms) function; it just guarantees that a certain amount of time has passed, without regard to *absolute* time. Note that both of these timers work by *adding* activity to the loop. That is, the loop can't go on to the next cycle until everything in the loop has finished, and that includes the timer.



**Figure 5.2** Wait Until Next ms Multiple (the two left loops) keeps things in sync. The right loop runs every 100 ms, but may be out of phase with the others.

Since LabVIEW executes everything inside the loop in parallel, the timer is presumably started at the same time as everything else. Note that “everything else” must be completed in *less* time than the desired interval. The timer does not have a magic ability to speed up or abort other functions. Wait Till Next ms Multiple will add on extra delay as necessary to make the tick count come out even, but not if the other tasks overrun the desired interval.

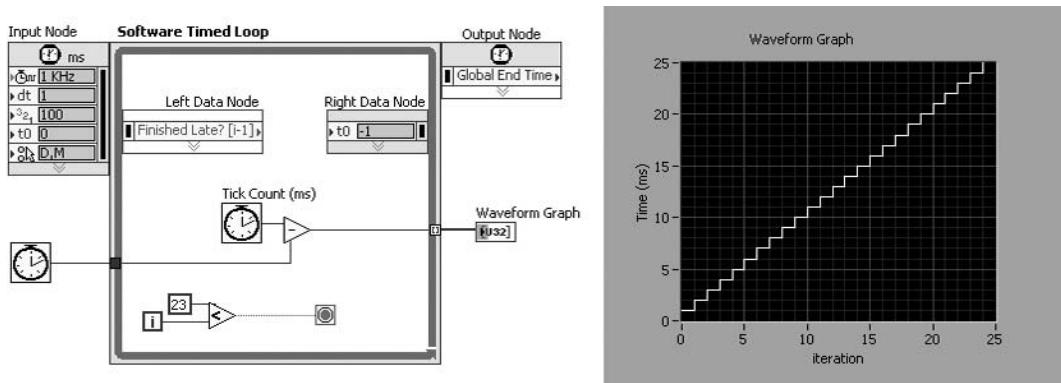
Both of these loop timers are asynchronous; that is, they do not tie up the entire machine while waiting for time to pass. Any time a call to Wait (ms) is encountered, LabVIEW puts the calling VI into a *scheduler queue* until the timer expires. If you want your loop to run as fast as possible but not hog all the CPU, place a Wait (ms) with 0 ms to wait. The 0-ms wait is a special instruction to the LabVIEW scheduler to let other waiting tasks run, but to put yours back in the schedule as soon as possible. This guarantees that other VIs have a chance to run, but still executes your loop as fast as possible. *Rule:* *Always put a timer in every loop structure.* Otherwise, the loop may lock out all other VIs, and even interfere with the responsiveness of the LabVIEW user interface. The exception to this rule occurs when you know there is some other activity in the loop that will periodically cause LabVIEW to put that part of the diagram in the scheduler queue. For instance, file I/O and many DAQ and GPIB operations have hidden delays or other scheduling behavior that meets this requirement.

## Timed structures

### CLAD

There are three programming structures in LabVIEW 8 with built-in scheduling behavior: a **Timed Loop**, a **Timed Sequence**, and a combination of the two called a **Timed Loop with Frames**. Timed Loops are most useful when you have multiple parallel loops you need to execute in a deterministic order. LabVIEW’s default method of executing code is multiple threads of time-sliced multitasking execution engines with a dash of priorities. In other words, your code will appear to run in parallel, but it is really sharing CPU time with the rest of your application (and everything else on your computer). Deterministically controlling the execution order of one loop over another is an incredibly difficult task—until now.

Timed Loops are as easy to use as a While Loop and ms Timer combination. The Timed Loop in Figure 5.3 executes every 1 ms and calculates the elapsed time. You configure the loop rate by adjusting the period, or dt. You can also specify an offset (t0) for when the loop should start. With these two settings you can configure multiple loops to run side by side with the same period but offset in time from each other. How to configure



**Figure 5.3** Timed Loops have a built-in scheduling mechanism. You can configure a Timed Loop's behavior dynamically at run time through terminals on the Input node, or statically at edit time through a configuration dialog.

the period and offset is all that most of us need to know about Timed Loops, but there is much more. You can change everything about a Timed structure's execution, except the name and the timing source, on the fly from within the structure. You can use the execution feedback from the Left Data Node in a process control loop to tell whether the structure started on time or finished late in the previous execution. Time-critical processes can dynamically tune the Timed structure's execution time and priority according to how timely it executed last time. The left data node even provides nanosecond-resolution timing information you can use to benchmark performance.

## Timing sources

Timed structures (loops and sequences) run off a configurable timing source. Figure 5.4 shows a typical configuration dialog. The default source is a 1-kHz clock (1-ms timer) on a PC. On an RT system with a Pentium controller, a 1-MHz clock is available, providing a 1- $\mu$ s timer. Other configuration options are the Period and Priority. In our dialog the loop is set to execute every 1000 ms by using the PC's internal 1-kHz clock. Note that you can also use a timing source connected to the structure's terminal. The timing source can be based on your computer's internal timer or linked through DAQmx to a hardware-timed DAQ event with **DAQmx Create Timing Source.vi** (Figure 5.5). You can even chain multiple timing sources together by using **Build Timing Source Hierarchy.vi**. One example in which multiple timing sources could come in handy is if you wanted a loop to execute based on time or every time a trigger occurred.

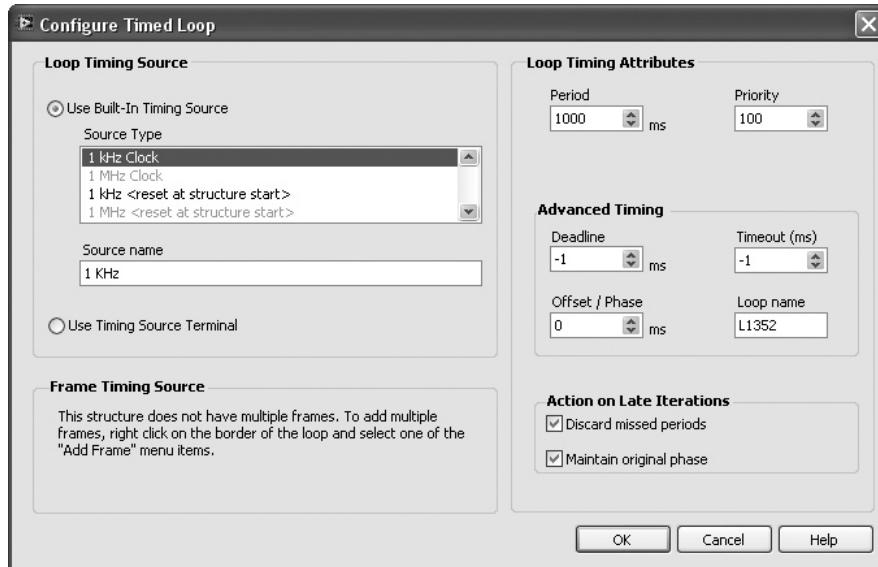


Figure 5.4 Configuration dialog for a Timed Loop with a single frame.

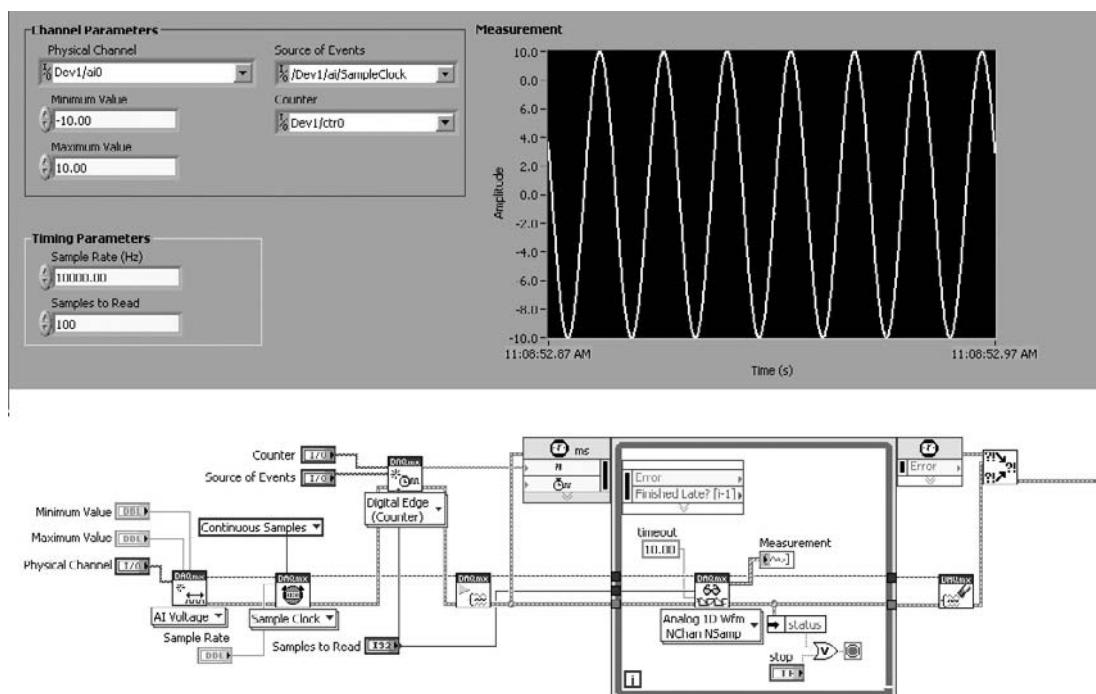


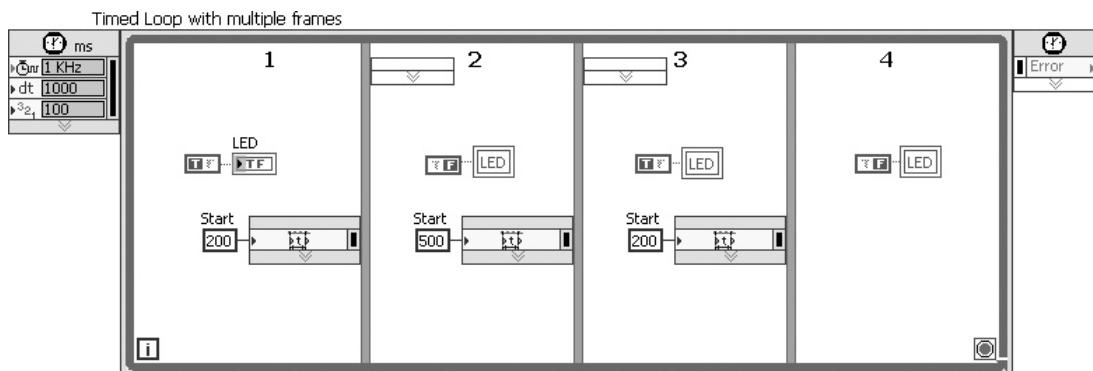
Figure 5.5 Timed structures can be synchronized to external timing sources with DAQmx.

It's important to note that once a timing source is started, it will continue to run even if the timed loop is not running! This can cause unexpected consequences when your Timed Loop starts running as fast as possible to try to catch up to its timing source. To avoid this, use the “<reset at structure start>” feature. This is important if you use a Timed Loop inside another loop. The loop needs to either reset its clock when it starts or be set to discard missed periods. Another method is to clear the timer after exiting the loop, by using **Clear Timing Source.vi**. This removes the timer but adds unnecessary overhead each time the timer has to be recreated.

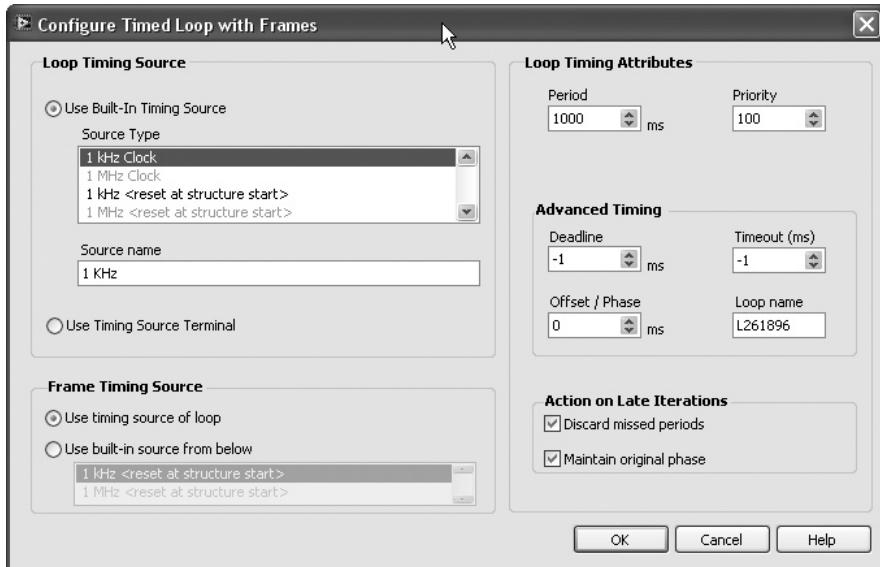
Timed Loops with Frames and Timed sequence structures have the unique capability of running off two timing sources. One source triggers the start of the loop, and a second frame timing source governs each frame. The example in Figure 5.6 shows a multiframe Timed Loop with dead time inserted between each frame. Figure 5.7 shows the configuration dialog.

### Execution and priority

Each Timed structure runs in its own thread at a priority level just beneath Time Critical and above High Priority. Timed Loops do not inherit the priority or execution system of the VI they run in. This means that a Timed structure will stop any other code from running until the structure completes. LabVIEW's Timed structure scheduler runs behind the scenes and controls execution based on each Timed structure's schedule and individual priority. Priorities are set before the Timed Loop executes, through either the configuration dialog or a terminal on the Input node. Priority can be adjusted inside the Timed



**Figure 5.6** Frames in a multiframe Timed structure can have start times relative to the completion of the previous frame. The effect is to put dead time between each frame. Frame 2 starts 200 ms after frame 1 completes, frame 3 starts 500 ms after frame 2, and frame 4 starts 200 ms after frame 3.



**Figure 5.7** Multiframe structures can have two timing sources.

Loop through the Right Data node. The Timed Loop's priority is a positive integer between 1 and 2,147,480,000. But don't get too excited since the maximum number of Timed structures you can have in memory, running or not, is 128.

Timed structures at the same priority do not multitask between each other. The scheduler is preemptive, but not multitasking. If two structures have the same priority, then dataflow determines which one executes first. Whichever structure starts execution first will finish before the other structure can start. Each Timed Loop runs to completion unless preempted by a higher-priority Timed Loop, a VI running at Time Critical priority, or the operating system. It is this last condition that kills any hope of deterministic operation on a desktop operating system.

On a real-time operating system, Timed structures are deterministic, meaning the timing can be predicted. This is crucial on an RTOS; but on a non-real-time operating system (such as Windows), Timed structures are not deterministic. This doesn't mean that you can't use Timed structures on Windows; just be aware that the OS can preempt your structure at any time to let a lower-priority task run. A real-time operating system runs code based on priority (higher-priority threads preempt lower-priority threads) while a desktop OS executes code based on "fairness"; each thread gets an opportunity to run. This is great

when you want to check e-mail in the background on your desktop, but terrible for a process controller. If you need deterministic execution, then use Timed structures on real-time systems such as LabVIEW RT. They are powerful and easy to use.

### Timing guidelines

All LabVIEW's platforms have built-in **multithreading** and **preemptive multitasking** that help to mitigate the effects of errant code consuming all the CPU cycles. Even if you forget to put in a timer and start running a tight little loop, every desktop operating system will eventually decide to put LabVIEW to sleep and to let other programs run for a while. But rest assured that the overall responsiveness of your computer will suffer, so *please* put timers in all your loops. For more information on how the LabVIEW execution system works, see Application Note 114, *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability*.

Here's when to use each of the timers:

- Highly regular loop timing—use Wait Until Next ms Multiple or a Timed Loop.
- Many parallel loops with regular timing—use Wait Until Next ms Multiple or Timed Loops.
- Arbitrary, asynchronous time delay to give other tasks some time to execute—use Wait (ms).
- Single-shot delays (as opposed to cyclic operations, such as loops)—use Wait (ms) or a Timed Sequence.

#### CLAD

**Absolute timing functions.** LabVIEW has a set of timing functions that deal with absolute time, which generally includes the date in addition to the time of day. When we are dealing with absolute time, the first thing we need to do is to establish our calendar's origin. While every computer platform uses a slightly different standard origin, the LabVIEW designers have attempted to give you a common interface. They set the origin as a time-zone-independent number of seconds that have elapsed since 12:00 A.M., Friday, January 1, 1904, Coordinated Universal Time (UTC). For convenience, we call this **epoch** time. On your computer, you must configure your time zone (typically through a control panel setting) to shift the hours to your part of the globe. Internally LabVIEW calculates time based on UTC, but displays time according to local time. With LabVIEW 8 you finally have the ability to specify whether time is UTC or local. Prior to LabVIEW 8, coordinating time on multiple computers across the Internet was not straightforward.

Now you can schedule an event on multiple computers to UTC without worrying about time zone offset or daylight savings time.

Epoch time values are available in two formats: as a **timestamp** and as a cluster known as a **date time rec**. Timestamp is a high-resolution 128-bit data type capable of handling dates between 1600 A.D. and 3000 A.D. The upper **quad int** (64 bits) is the number of seconds since January 1, 1904, and the lower 64 bits is the fractional second. The date/time rec cluster contains nine I32 values (second, minute, hour, day, etc.) and one DBL (fractional second). The seconds value is just what we've discussed: the number of seconds since the beginning of 1904. The date/time rec cluster is useful for extracting individual time and date values, such as finding the current month.

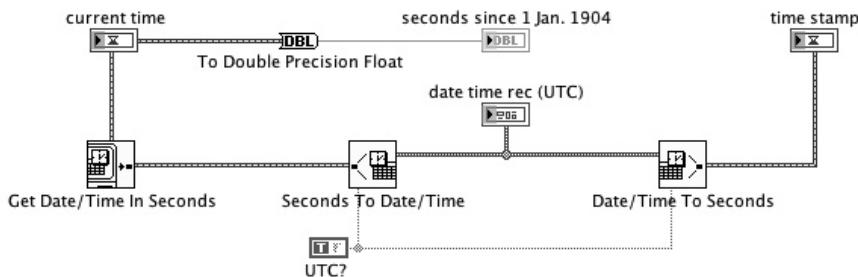
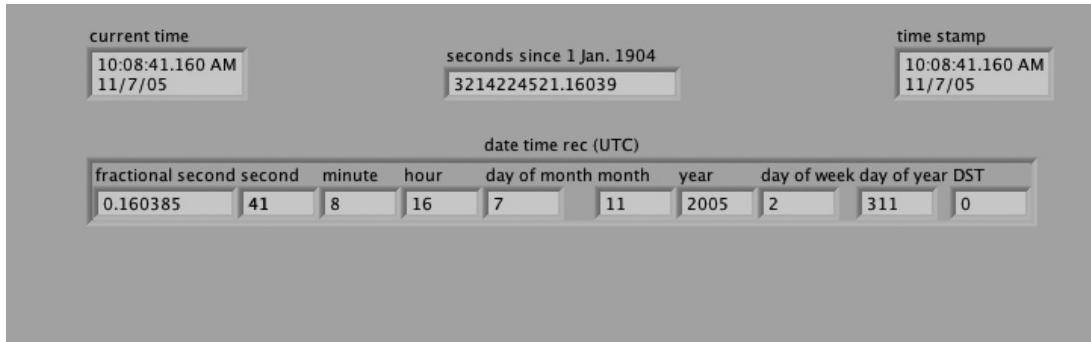
Here is a quick introduction to the absolute time functions and their uses:

- **Get Date/Time In Seconds** returns a timestamp, which can be converted into the epoch seconds as a DBL float.
- **Get Date/Time String** returns two separate formatted strings containing time and date. It's nice for screen displays or printing in reports and has several formatting options.
- **Format Date/Time String** returns a single formatted string containing time and date. You supply a formatting string that allows you to extract any or all possible components of date and time. For instance, the format string `%H:%M:%S%3u` asks for hours, minutes, and fractional seconds with three decimal digits with colons between the values. A typical output string would look like this: `15:38:17.736`
- **Seconds To Date/Time** returns a date/time rec cluster. You might use this function when you want to do computations based on a particular aspect of date or time.
- **Date/Time To Seconds** converts the date/time rec back to seconds. It's useful when you want to combine individual time and date items, such as letting the user sift through data by the day of the week.

Figure 5.8 shows how many of these functions can be connected.

#### Sending timing data to other applications

Many graphing and spreadsheet programs do a lousy job of importing timestamps of the form `23-Jun-1990 10:03:17`. Epoch seconds are OK, but then the other program has to know how to compute the date and time, which requires a big algorithm. Most spreadsheet programs can



**Figure 5.8** How to switch between the various formats for system time, particularly getting in and out of the date/time rec cluster.

handle epoch seconds if you give them a little help. In Microsoft Excel, for instance, you must divide the epoch seconds value by 86,400, which is the number of seconds in 1 day. The result is what Excel calls a *Serial Number*, where the integer part is the number of days since the zero year and the fractional part is a fraction of 1 day. Then you just format the number as date and time. Again, watch out for the numeric precision problem if you're importing epoch seconds.

### High-resolution and high-accuracy timing

If your application requires higher accuracy or resolution than the built-in software timing functions can supply, then you will have to use a hardware solution. The first thing to do is to consider your specifications.

Some experiments simply need a long-term stable time base, over scales of days or even years, to synchronize with other events on other systems. Or you might be concerned with measuring time intervals in the subsecond range where short-term accuracy is most important. These specifications will influence your choice of timing hardware.

For long-term stability, you might get away with setting your computer's clock with one of the network time references; such options are built into all modern operating systems. They guarantee NIST-traceable long-term stability, but there is, of course, unknown latency in the networking that creates uncertainty at the moment your clock is updated.

Excellent time accuracy can be obtained by locking to signals received from the Global Positioning System (GPS) satellites, which provide a 1-Hz clock with an uncertainty of less than 100 ns and essentially no long-term drift. The key to using GPS timing sources is to obtain a suitable interface to your computer and perhaps a LabVIEW driver. One GPS board we like is the TSAT cPCI from KSI. It has a very complete LabVIEW driver with some extra features such as the ability to programmatically toggle a TTL line when the GPS time matches an internal register. You can use this programmable Match output to hardware-gate—a data acquisition task. Keep in mind the ultimate limitation for precision timing applications: software latency. It takes a finite amount of time to call the program that fetches the time measurement or that triggers the timing hardware. If you can set up the hardware in such a way that there is no software “in the loop,” very high precision is feasible. We placed TSAT cPCI cards into multiple remote PXI data acquisition systems and were able to synchronize data collection with close to 100 ns of accuracy. This was essential for our application, but a GPS board is probably overkill for most applications.

Perhaps the easiest way to gain timing resolution is to use a National Instruments data acquisition (DAQ) board as a timekeeper. All DAQ boards include a reasonably stable crystal oscillator time base with microsecond (or better) resolution that is used to time A/D and D/A conversions and to drive various timers. Through the DAQ VI library, you can use these timers to regulate the cycle time of software loops, or you can use them as high-resolution clocks for timing short-term events. This DAQ solution is particularly good with the onboard counter/timers for applications such as frequency counting and time interval measurement.

But what if the crystal clock on your DAQ board is not good enough? Looking in the National Instruments catalog at the specifications for most of the DAQ boards, we find that the absolute accuracy is  $\pm 0.01$  percent, or  $\pm 100$  ppm, with no mention of the temperature coefficient. Is that sufficient for your application? For high-quality calibration work and precision measurements, probably not. It might even be a problem for frequency determination using spectral estimation. In the LabVIEW analysis library, there's a VI called Extract Single Tone Information that locates, with very high precision, the fundamental frequency of a sampled waveform. If the noise level is low, this VI can

return a frequency estimate that is accurate to better than 1 ppm—but only if the sample clock is that accurate.

National Instruments does have higher-accuracy boards. In the counter/timer line, the 6608 counter/timer module has an oven-controlled crystal oscillator (OCXO) that can be adjusted to match an external reference, and it has a temperature coefficient of  $1 \times 10^{-11}/^{\circ}\text{C}$  and aging of  $1 \times 10^{-7}$  per year. With all the National Instruments clock generators, you typically connect them to your DAQ board via real-time system integration (RTSI) bus signals or via programmable function input (PFI) lines. If you're using an external clock generator, the PFI inputs are the only way. There are several DAQ example VIs that show you how to use external clocking for analog and digital I/O operations.

A bargain in the world of stable time bases is the Stanford Research PRS10 rubidium frequency standard. For about US\$1500, you get a 10-MHz oscillator that fits in the palm of your hand with a short-term stability of  $1 \times 10^{-11}$  over 1 s, aging of  $5 \times 10^{-10}$  per year, and a temperature coefficient of  $5 \times 10^{-11}/^{\circ}\text{C}$ . It also has the lowest phase noise of any oscillator at any price, which is important if you are phase lock-multiplying the time base to obtain microwave frequencies. It can be locked to a 1-Hz GPS clock to enhance its long-term stability.

If your facility is set up for it, you can also read the time from time-base standards such as IRIG (Inter-Range Instrumentation Group) that use a serial interface with up to 100-ns resolution. Interface boards for IRIG (and GPS) are available from Bancomm (a division of Dataum Corporation) and from TrueTime. Both manufacturers offer plug-in boards that keep your computer's clock/calendar synchronized, as well as providing very accurate digital timing outputs. Some LabVIEW drivers are available, or you can use the shared libraries (DLLs) or C-callable routines to make your own.

## Bibliography

Application Note 200, *Using the Timed Loop to Write Multirate Applications in LabVIEW*, [www.ni.com](http://www.ni.com), National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2004.

*This page intentionally left blank*

## Synchronization

When you build a LabVIEW application, you'll eventually find the need to go beyond simple sequencing and looping to handle all the things that are going on in the world outside your computer. A big step in programming complexity appears when you start handling **events**. An event is usually defined as something external to your program that says, "Hey, I need service!" and demands that service in a timely fashion. A simple example is a user clicking a button on the panel to start or stop an activity. Other kinds of events come from hardware, such as a GPIB service request (SRQ) or a message coming in from a network connection. Clearly, these random events don't fall within the simple boundaries of sequential programming.

There are lots of ways to handle events in LabVIEW, and many of them involve creating parallel paths of execution where each path is responsible for a sequential or nonsequential (event-driven) activity.

Since G is intrinsically parallel, you can create an unlimited number of parallel loops or VIs and have confidence that they'll run with some kind of timesharing. But how do you handle those "simultaneous" events in such a way as to avoid collisions and misordering of execution when it matters? And what about passing information between loops or VIs? The complexity and the list of questions just seem to grow and grow.

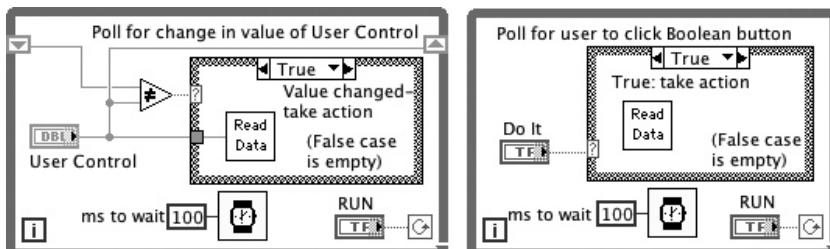
LabVIEW has some powerful synchronization features, and there are standard techniques that can help you deal with the challenges of parallel programming and event handling. By the way, this is an advanced topic, so if some of the material here seems strange, it's because you might not need many of these features very often (we'll start with the

easier ones, so even you beginners can keep reading). But once you see how these features work, you'll soon become an armed and dangerous *LabVIEW dude*.\*

## Polling

Let's start off with a straightforward event-handling technique that does not use any esoteric functions or tricks: **polling**. When you are polling, you periodically check the status of something (a **flag**), looking for a particular value or change of value. Polling is very simple to program (Figure 6.1), is easy to understand and debug, is very widely used (even by highly experienced LabVIEW dudes!), and is acceptable practice for thousands of applications in all programming languages. The main drawback to polling is that it adds some overhead because of the repeated testing and looping. (If you do it wrong, it can add *lots* of overhead.)

In the left example in Figure 6.1, we're doing a very simple computation—the comparison of two values—and then deciding whether to take action. There's also a timer in the loop that guarantees that the computation will be done only 10 times per second. Now that's pretty low overhead. But what if we left the timer out? Then LabVIEW would attempt to run the loop as fast as possible—and that could be millions of times per second—and it might hog the CPU. Similarly, you might accidentally insert some complex calculation in the loop where it's not needed, and each cycle of the polling loop would then be more costly.



**Figure 6.1** A simple polling loop handles a user-interface event. The left example watches for a change in a numeric control. The right example awaits a True value from the Boolean button.

---

\*There really is an official LabVIEW Dude. On the registry system at Lawrence Livermore National Lab, Gary got tired of getting confused with the other two Gary Johnsons at the Lab, so he made his official nickname "LabVIEW Dude." It was peculiar, but effective. Just like Gary.

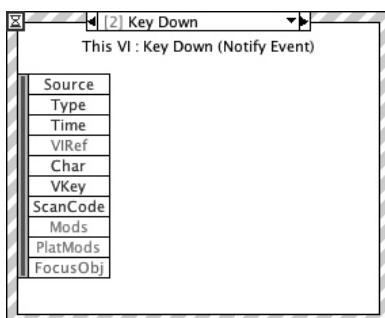
The other drawback to polling is that you only check the flag's state every so often. If you're trying to handle many events in rapid succession, a polling scheme can be overrun and miss some events. An example might be trying to handle individual bits of data coming in through a high-speed data port. In such cases, there is often a hardware solution (such as the hardware buffers on your serial port interface) or a lower-level driver solution (such as the serial port handler in your operating system). With these helpers in the background doing the fast stuff, your LabVIEW program can often revert to polling for larger packets of data that don't come so often.

## Events

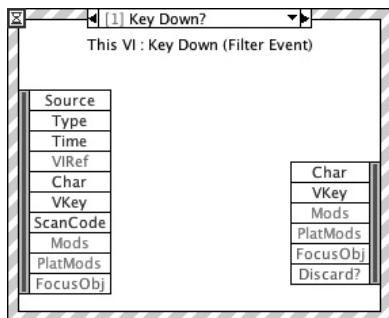
### CLAD

Wouldn't it be great if we had a way to handle user-interface events without polling? We do and it's called the **Event** structure. Remember, an *event* is a notification that something has happened. The Event structure bundles handling of user-interface events (or notifications) into one structure without the need for polling front panel controls, and with a lot fewer wires! The Event structure sleeps without consuming CPU cycles until an event occurs, and when it wakes up, it automatically executes the correct Event case. The Event structure doesn't miss any events and handles all events in the order in which they happen.

The Event structure looks a lot like a Case structure. You can add, edit, or delete events through the Event structure's pop-up menu. Events are broken down into two types: **Notify** events and **Filter** events. A Notify event lets you know that something happened and LabVIEW has handled it. Pressing a key on the keyboard can trigger an event. As a notify event, LabVIEW lets you know the key was pressed so you can use it in your block diagram, but all you can do is react to the event. Figure 6.2 shows a key down Notify event. A Filter



**Figure 6.2** Notify event. Configured to trigger a notification when any key is pressed.



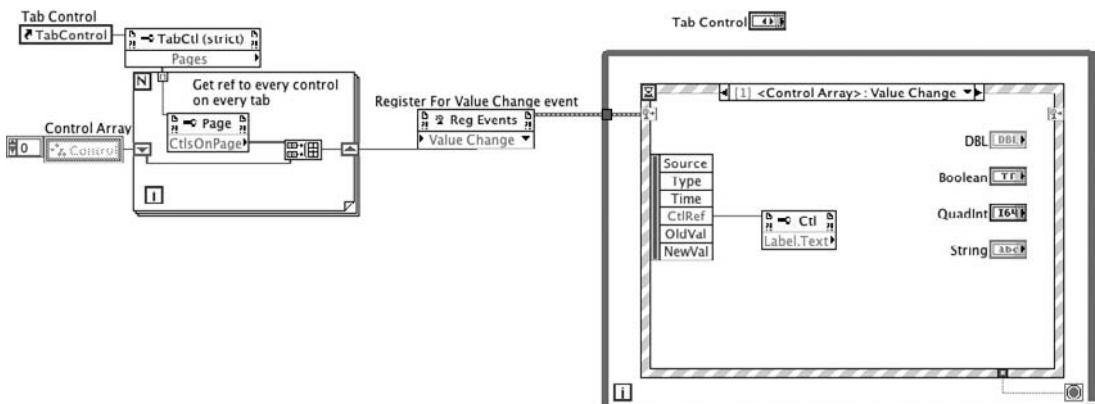
**Figure 6.3** Filter event. This triggers a notification when any key is pressed, but the key press can be modified or discarded.

event (Figure 6.3) allows you to change the event's data as it happens or even discard the event. Filter events have the same name as Notify events, but end with a question mark (“?”).

The Event structure can save a lot of time and energy, but it can also drive you crazy if you aren't alert. The default configuration of all events is to lock the front panel until the Event case has finished execution. If you put any time-consuming processing inside the Event structure, your front panel will lock up until the event completes. *Rule: Handle all but the most basic processing external to the Event structure.* Later in this chapter we'll show you a powerful design pattern you can use to pass commands from an event loop to a parallel processing loop with queues.

Use events to enhance dataflow, not bypass it. This is especially true with the mechanical action of booleans. A latching boolean does not reset until the block diagram has read its value. If your latching boolean is outside floating on the block diagram, it will never reset. Be sure to place any latching booleans inside their Event case so their mechanical action will be correct. Changing a control's value through a local variable will not generate an event – use the property “Value (Signaling)” to trigger an event.

The advanced topic of **Dynamic Events** allows you to define and dynamically register your own unique events and pass them between subdiagrams. Dynamic events can unnecessarily complicate your application, so use them only when needed and make sure to document what you are doing and why. Figure 6.4 illustrates configuring a dynamic event to trigger a single Event case for multiple controls. This eliminates manually configuring an Event case for each control—a tedious task if you have tens, or even hundreds, of controls. Property nodes are used to get the control reference to every control on every



**Figure 6.4** Use control references for dynamic event registration. A single Event case handles multiple Value Change events.

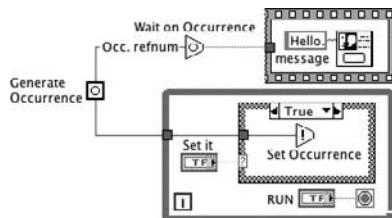
page of a tab control. The 1D array of references is then used to register all the controls for a value change event. Whenever any control's value is changed, a single Event case will be fired. Inside the Event case we have a reference to the control and its data value as a variant. Also note that all the controls are inside the Event case.

## Occurrences

An **occurrence** is a synchronization mechanism that allows parallel parts of a LabVIEW program to notify each other when an event has occurred. It's a kind of software trigger. The main reason for using occurrences is to avoid polling and thus reduce overhead. An occurrence does not use any CPU cycles while waiting. For general use, you begin by calling the **Generate Occurrence** function to create an occurrence refnum that must be passed to all other occurrence operations.

Then you can either wait for an occurrence to happen by calling the **Wait On Occurrence** function or use **Set Occurrence** to make an event happen. Any number of Wait On Occurrence nodes can exist within your LabVIEW environment, and all will be triggered simultaneously when the associated Set Occurrence function is called.

Figure 6.5 is a very simple demonstration of occurrences. First, we generate a new occurrence refnum and pass that to the Wait and Set functions that we wish to have related to one another. Next, there's our old friend the polling loop that waits for the user to click the Do it button. When that happens, the occurrence is set and, as if by magic, the event triggers the Wait function, and a friendly dialog box pops up. The magic here is the transmission of information (an event) through thin



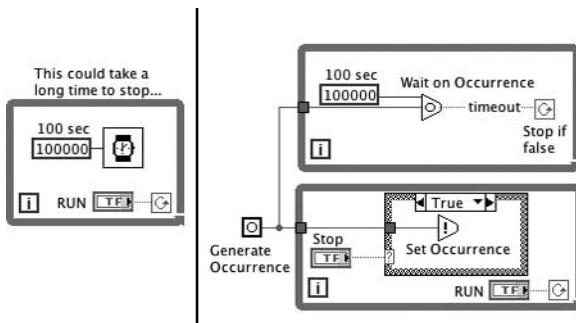
**Figure 6.5** A trivial occurrence demonstration. The lower loop sets the occurrence when the user clicks the Set It button, causing the Wait On Occurrence function to trigger the dialog box. Terminating the While Loop without setting the occurrence causes the Wait On Occurrence function to wait forever.

air from one part of the diagram to another. In fact, the Wait function didn't even have to be on the same piece of diagram; it could be out someplace in another VI. All it has to have is the appropriate occurrence refnum.

It appears that, as with global variables, we have a new way to violate dataflow programming, hide relationships among elements in our programs, and make things occur for no apparent reason in unrelated parts of our application. Indeed we do, and that's why you must always use occurrences with caution and reservation. As soon as the Set and Wait functions become dislocated, you need to type in some comments explaining what is going on.

Well, things seem pretty reasonable so far, but there is much more to this occurrence game. First, there is a **ms time-out** input on the Wait On Occurrence function that determines how long it will wait before it gets bored and gives up. The default time-out value, -1, will cause it to wait forever until the occurrence is set. If it times out, its output is set to True. This can be a safety mechanism of a sort, when you're not sure if you'll ever get around to setting the occurrence and you want a process to proceed after a while. A very useful application for the time-out is an **abortable wait**, first introduced by Lynda Gruggett (*LabVIEW Technical Resource*, vol. 3, no. 2). Imagine that you have a While Loop that has to cycle every 100 s, as shown in the left side of Figure 6.6. If you try to stop the loop by setting the conditional terminal to False, you may have to wait a long time. Instead, try the abortable wait (Figure 6.6, right).

To implement the abortable wait, the loop you want to time and abort is configured with a Wait On Occurrence wired to a time-out value. Whenever the time-out period has expired, the Wait returns True, and the While Loop cycles again, only to be suspended once again on the Wait.



**Figure 6.6** An abortable wait (right) can be created from a Wait On Occurrence function by using the time-out feature.

When the occurrence is set (by the lower loop, in this example), the Wait is instantly triggered and returns False for the time-out flag, and its loop immediately stops.

Now for the trickiest feature of the Wait On Occurrence function: the **ignore previous** input. It's basically there to allow you to determine whether to discard occurrence events from a previous execution of the VI that it's in. Actually, it's more complicated than that, so here is a painfully detailed explanation in case you desperately want to know all the gory details.

Each Wait On Occurrence function “remembers” what occurrence it last waited on and at what time it continued (either because the occurrence triggered or because of a time-out). When a VI is loaded, each

Wait On Occurrence is initialized with a nonexisting occurrence. When a Wait On Occurrence is called and *ignore previous* is *False*, there are four potential outcomes:

1. The occurrence has *never* been set. In this case Wait On Occurrence simply waits.
2. The occurrence *has* been set since this Wait On Occurrence last executed. In this case Wait On Occurrence does not wait.
3. The occurrence was last set before this Wait last executed, *and* last time this Wait was called it waited on the *same* occurrence. Wait On Occurrence will then wait.
4. The occurrence was last set before this Wait last executed, *but* last time this Wait was called it waited on a *different* occurrence. In this case Wait will *not* wait!

The first three cases are pretty clear, but the last one may seem a bit strange. It will arise only if you have a Wait On Occurrence inside a

loop (or inside a *reentrant* VI in a loop) and it waits on *different* occurrences (out of an array, for example). This can also arise if it is inside a non-reentrant VI and the VI is called with different occurrences. Fortunately, these obscure cases generally don't happen.

Wait On Occurrence behaves in this way due to its implementation. Each occurrence "knows" the last time it was triggered, and each Wait On Occurrence remembers the occurrence it was last called with and what time it triggered (or timed out). When Wait On Occurrence is called and ignore previous is False, it will look at its input. If the input is the same occurrence as last time, it will look at the time of the last firing and wait depending on whether the time was later than the last execution. If the input is *not* the same as last time, it will simply look at the time and wait depending on whether it has ever been fired.

After you've tried a few things with occurrences, you'll get the hang of it and come up with some practical uses. Some experienced LabVIEW users have written very elaborate scheduling and timing programs based on occurrences (far too complex to document here). In essence, you use sets of timers and various logical conditions to determine when occurrences are to be triggered. All your tasks that do the real work reside in loops that wait on those occurrences. This gives you centralized control over the execution of a large number of tasks. It's a great way to solve a complex process control situation. Just hope you don't have to explain it to a beginner.

To sum up our section on occurrences, here is a quote from Stepan Rhia, a former top-notch LabVIEW developer at National Instruments:

One may think that occurrences are quirky, a pain to use, and that they should be avoided. One might be right! They are very low level and you often have to add functionality to them in order to use them effectively. In other words, they are not for the faint of heart. Anything implemented with occurrences can also be implemented without them, but maybe not as efficiently.

## Notifiers

### CLAD

As Stepan Rhia indicated, adding functionality around occurrences can make them more useful. And that's exactly what a **notifier** does: It gives you a way to send information along with the occurrence trigger.

Although notifiers still have some of the hazards of occurrences (obscuring the flow of data), they do a good job of hiding the low-level trickery.

To use a notifier, follow these general steps on your diagram. First, use the **Create Notifier** VI to get a notifier refnum that you pass to your other notifier VIs. Second, place a **Wait On Notification** VI

in a loop or wherever you want your program to be suspended until the notification event arrives along with its data. Third, call the **Send Notification** VI to generate the event and send the data. When you're all done, call **Destroy Notifier** to clean up.

Here's an example that uses notifiers to pass data when an alarm limit is exceeded. We'll follow the numbered zones in Figure 6.7 to see how it works.

1. A new notifier is created. Its refnum is passed to two While Loops.
2. The two loops don't have to reside on the same diagram; they just need to have the refnum passed to them, perhaps through a global variable.
3. Inside the True frame of the Case structure, the value is submitted to the Send Notification VI. Behind the scenes, that VI generates an occurrence and queues up the data.
4. The lower While Loop is waiting for the notification to be set. When it arrives, the Wait On Notification VI returns the data.
5. A message for the user is assembled, and a dialog box pops up.
6. The upper loop stops when the user clicks the Stop button, and the notifier is destroyed.
7. The lower loop stops when the Wait On Notification returns an error because the notifier no longer exists.

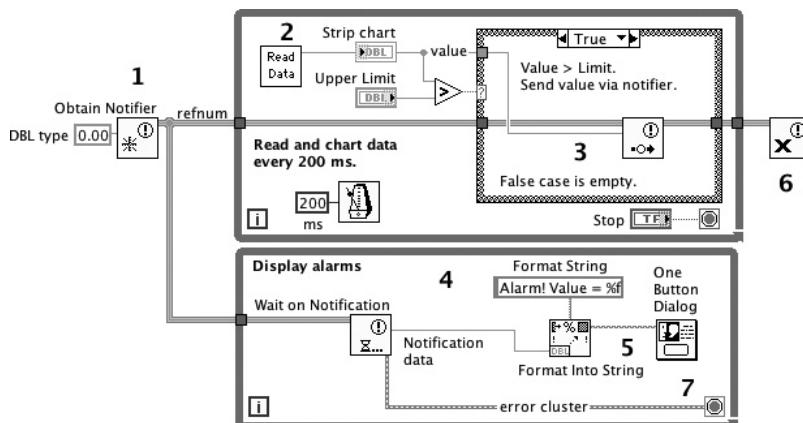


Figure 6.7 A notifier sends data from the upper loop to the lower one when an alarm limit is exceeded.

If you actually create and run this VI, you'll find one misbehavior. When the notifier is destroyed and the bottom loop terminates, the Wait On Notification VI returns an empty number, which is passed to the dialog box to be displayed as a value of 0; it's a false alarm. To eliminate this bug, all nodes that use the data string should be enclosed in a Case structure that tests the error cluster coming from the Wait On Notification VI. In the remainder of this chapter, you'll see that technique used as a standard practice.

Wait On Notification includes the time-out. Ignore previous features of Wait On Occurrence, and they work the same way. However, most of the time you don't need to worry about them. The only trick you have to remember is to include a way to terminate any loops that are waiting. The error test scheme used here is pretty simple. Alternatively, you could send a notification that contains a message explicitly telling the recipient to shut down.

One important point to keep in mind about notifiers is that the data buffer is only one element deep. Although you can use a single notifier to pass data one way to multiple loops, any loop running slower than the notification will miss data.

## Queues



**Queues** function a lot like notifiers but store data in a FIFO (first in, first out) buffer. The important differences between queues and notifiers are as follows:

1. A notifier can pass the same data to multiple loops, but a notifier's data buffer is only one element deep. All notifier's writes are destructive.
2. A queue can have an infinitely deep data buffer (within the limits of your machine), but it's difficult to pass the data to more than one loop. A queue's reads are destructive.

Another important difference between queues and notifiers is the way they behave if no one reads the data. You might think that a queue with a buffer size of 1 would behave the same as a notifier. But once a fixed-size queue is filled, the **producer** loop either can time out and not insert the data, or can run at the same rate as the **consumer**.

Figure 6.8 shows the alarm notification application of Figure 6.7 rewritten using queues. Note that the application looks and acts the same, but with one exception: Because the queue size is infinite, no alarms will be missed if the user fails to press the acknowledgment button in time. Every alarm will be stored in the queue until dequeued.

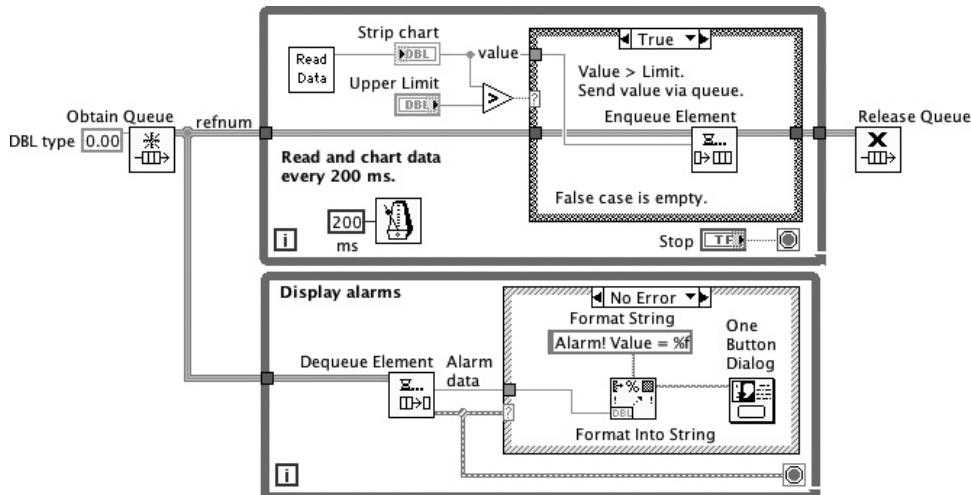


Figure 6.8 Alarm notification using queues. No alarms will be dropped because of user inactivity.

Use a **queue** when you can't afford to miss any data passed between parallel loops.

The asynchronous nature of queues makes for a great way to pass messages between two loops running at different rates. Figure 6.9 shows an implementation of this using an Event structure in one loop and a message-driven state machine in the other. The **Queued Message Handler** is one of the most powerful design patterns in LabVIEW for handling user-interface-driven applications. Notice we're giving the queue a unique name, to avoid **namespace conflicts**. All we need to insert and remove data from a queue is the queue refnum or its name. If we have another queue in memory using a string data type, the two queues will use the same data buffer. This can be hard to troubleshoot!

*Rule: Give each queue a unique name at creation.*

## Semaphores

### CLAD

With parallel execution, sometimes you find a **shared resource** called by several VIs, and a whole new crop of problems pop up. A shared resource might be a global variable, a file, or a hardware driver that could suffer some kind of harm if it's accessed simultaneously or in an undesirable sequence by several VIs. These **critical sections** of your program can be guarded in several ways.

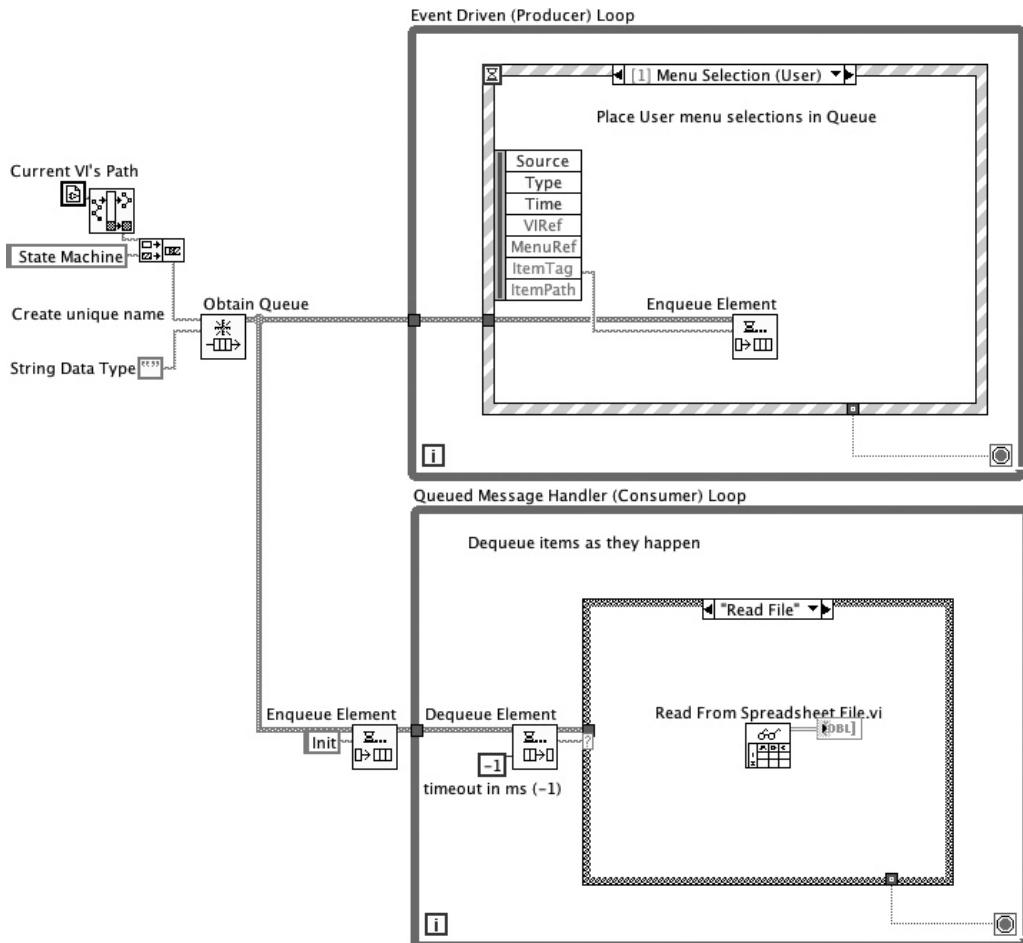


Figure 6.9 Queued Message Handler. Each event is placed in queue and handled in order by the consumer loop.

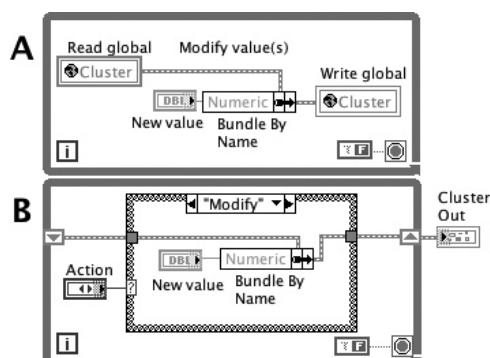
One way is to encapsulate the critical section of code in a subVI. The LabVIEW execution system says that a given subVI can only be running under the auspices of one calling VI, unless it is made **reentrant** in the VI Properties dialog. For instance, say that you have a scratch file that several VIs need to read or write. If the open-read/write-close sequence was independently placed in each of those VIs, it's possible that the file could be *simultaneously* written and/or read. What will happen then? A simple solution is to put the file access functions in a subVI. That way, only one caller can have access to the file at any moment.

Global variables are another kind of shared resource that LabVIEW programmers have trouble with. A familiar clash occurs when you're maintaining a kind of global database. Many VIs may need to access it, and the most hazardous moment comes during a read-modify-write cycle, much like the scratch file example. Here, the global variable is read, the data is modified, and then it is written back to the global (Figure 6.10A). What happens if several VIs attempt the same operation? Who will be the last to write the data? This is also known as a **race condition**.

While built-in LabVIEW globals have no protection, you can solve the problem by encapsulating them inside subVIs, which once again limits access to one caller at a time. Or you can use global variables that are based on shift registers (Figure 6.10B) and have all the programmed access there in a single subVI.

Computer science has given us a formal protection mechanism. A **semaphore**, also known as a **mutex** (short for *mutually exclusive*), is an object that protects access to a shared resource. To access a critical section, a task has to wait until the semaphore is available and then immediately set the semaphore to *busy*, thus locking out access by other tasks. It's also possible for semaphores to permit more than one task (up to a predefined limit) to be in a critical section.

A new semaphore is created with the **Create Semaphore VI**. It has a *size* input that determines how many different tasks can use the semaphore at the same time. Each time a task starts using the semaphore, the semaphore size is decremented. When the size reaches 0, any task trying to use the semaphore must wait until the semaphore is released by another task. If the size is set to 1 (the default), only one



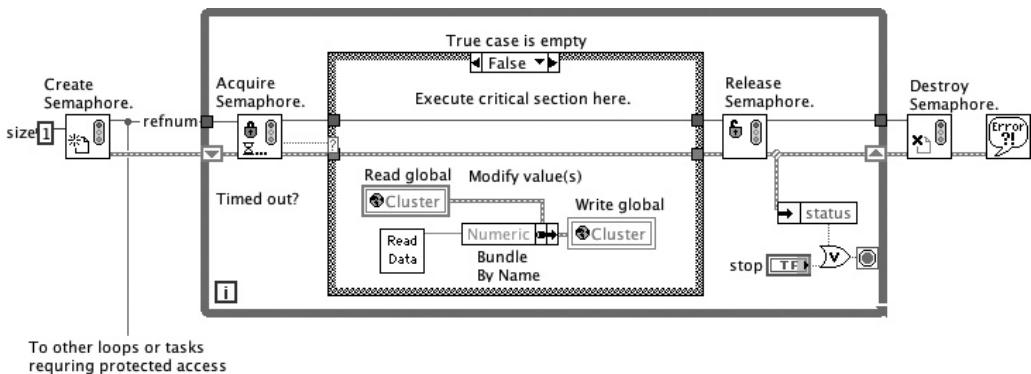
**Figure 6.10** Global variables are a classic risk for race conditions. You can protect such data by encapsulating it in a subVI or by storing the data in shift registers.

task can access the critical section. If you’re trying to avoid a race condition, use 1 as the size.

A task indicates that it wants to use a semaphore by calling the **Acquire Semaphore VI**. When the size of the semaphore is greater than 0, the VI immediately returns and the task proceeds. If the semaphore size is 0-, the task waits until the semaphore becomes available. There’s a timeout available in case you want to proceed even if the semaphore never becomes available. When a task successfully acquires a semaphore and is finished with its critical section, it releases the semaphore by calling the **Release Semaphore VI**. When a semaphore is no longer needed, call the **Destroy Semaphore VI**. If there are any Acquire Semaphore VIs waiting, they immediately time out and return an error.

Figure 6.11 is an example that uses a semaphore to protect access to a global variable. Only one loop is shown accessing the critical read-modify-write section, but there could be any number of them. Or the critical section could reside in another VI, with the semaphore refnum passed along in a global variable. This example follows the standard sequence of events: Create the semaphore, wait for it to become available, run the critical section, and then release the semaphore. Destroying the semaphore can tell all other users that it’s time to quit.

Semaphores are widely used in *multithreaded* operating systems where many tasks potentially require simultaneous access to a common resource. Think about your desktop computer’s disk system and what would happen if there were no coordination between the applications that needed to read and write data there. Since LabVIEW supports multithreaded execution of VIs, its execution system uses semaphores internally to avoid clashes when VIs need to use shared code. Where will *you* use semaphores?



**Figure 6.11** Semaphores provide a way to protect critical sections of code that can’t tolerate simultaneous access from parallel calling VIs.

## Me and You, Rendezvous

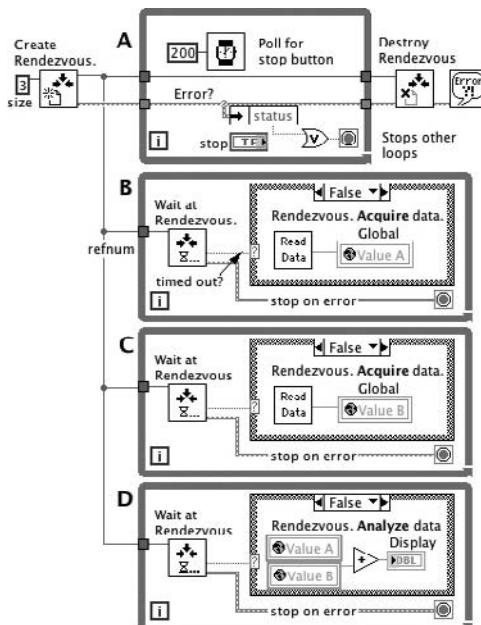
Here's another interesting situation that can arise with parallel tasks.

Say that you have two independent data acquisition tasks, perhaps using different hardware, each of which requires an unpredictable amount of time to complete an acquisition. (In other words, they're about as asynchronous as you can get.) But say that you also need to do some joint processing on each batch of data that the two tasks return; perhaps you need to combine their data into a single analysis and display routine. How do you get these things synchronized?

One way to synchronize such tasks is to use the **rendezvous** VIs. Each task that reaches the rendezvous point waits until a specified number of tasks are waiting, at which point all tasks proceed with execution. It's like a trigger that requires a unanimous vote from all the participants; nobody starts until we're *all* ready.

To arrange a rendezvous in LabVIEW, begin by creating a rendezvous refnum with the **Create Rendezvous** VI. Pass that refnum to all the required tasks, as usual. Use the **Wait At Rendezvous** VI to cause a loop to pause until everyone is ready to proceed. When you're all through, call the **Destroy Rendezvous** VI.

Figure 6.12 shows a solution to our data acquisition problem using rendezvous VIs. We're going to acquire data from two asynchronous



**Figure 6.12** Rendezvous VIs can synchronize parallel tasks by making them wait for each other.

sources and then analyze it. In this solution, there are four parallel While Loops that could just as well reside in separate VIs. (Leaving everything on one diagram indicates a much simpler way to solve this problem: Just put the two Read Data VIs in a single loop, and wire them up to the analysis section. Problem solved. Always look for the easiest solution!)

Start at the top of the diagram, while loop A is responsible for stopping the program. When that loop terminates, it destroys the rendezvous, which then stops the other parallel loops. Loops B and C asynchronously acquire data and store their respective measurements in global variables. Loop D reads the two global values and then does the analysis and display. The Wait On Rendezvous VIs suspend execution of all three loops until all are suspended. Note that the Create Rendezvous VI has a size parameter of 3 to arrange this condition. When the rendezvous condition is finally satisfied, loops B and C begin acquisition while loop D computes the result from the previous acquisition.

There are always other solutions to a given problem. Perhaps you could have the analysis loop trigger the acquisition VIs with occurrences, and then have the analysis loop wait for the results via a pair of notifiers. The acquisition loops would fire their respective notifiers when acquisition was complete, and the analysis loop could be arranged so that it waited for *both* notifiers before doing the computation. And there is no doubt a solution based on queues. Which way you solve your problem is dependent upon your skill and comfort level with each technique and the abilities of anyone else who needs to understand and perhaps modify your program.

---

Chapter  
7

# Files

Sooner, not later, you're going to be saving data in disk files for future analysis. Perhaps the files will be read by LabVIEW or another application on your computer or on another machine of different manufacture. In any case, that data is (hopefully) important to someone, so you need to study the techniques available in LabVIEW for getting the data on disk reliably and without too much grief.

Before you start shoveling data into files, make sure that you understand the requirements of the application(s) that will be reading your files. Every application has preferred formats that are described in the appropriate manuals. If all else fails, it's usually a safe bet to write out numbers as **ASCII text** files, but even that simple format can cause problems at import time—things including strange header information, incorrect combinations of carriage returns and/or line feeds, unequal column lengths, too many columns, or wrong numeric formats. **Binary** files are even worse, requiring tight specifications for both the writer and the reader. They are much faster to write or read and more compact than text files though, so learn to handle them as well. This chapter includes discussions of some common formats and techniques that you can use to handle them.

Study and understand your computer's file system. The online LabVIEW help discusses some important details such as path names and file reference numbers (refnums). If things really get gritty, you can also refer to the operating system reference manuals, or one of the many programming guides you can pick up at the bookstore.

## Accessing Files

File operations are a three-step process. First, you create or open a file. Second, you write data to the file and/or read data from the file. Third, you close the file. When creating or opening a file, you must

specify its location, or **path**. Modern computing systems employ a hierarchical file system, which imposes a directory structure on the storage medium. You store files inside directories, which can in turn contain other directories. To locate a file within the file system, LabVIEW uses a path-naming scheme that works consistently across all operating systems. On the Macintosh and Windows, you can have multiple drives attached to your machine, and each drive is explicitly referenced. On Sun and Linux, the physical implementation is hidden from you. Here are some examples of absolute path names:

#### Macintosh

HD80:My Data Folder:Data 123

#### Windows

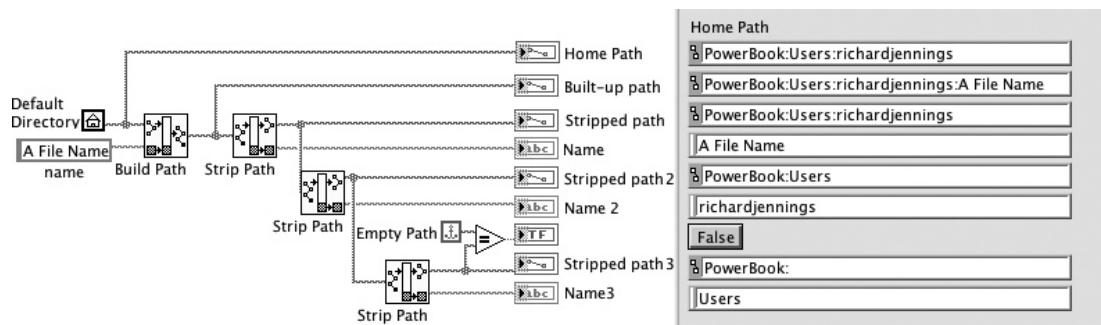
C:\JOE\PROGS\DATA\DATA123.DAT

#### UNIX

/usr/johnny/labview/examples/data\_123.dat

LabVIEW's **Path** control (from the String and Path palette) automatically checks the format of the path name that you enter and attempts to coerce it into something valid for your operating system.

Paths can be built and parsed, just as strings can. In fact, you can convert strings to paths and back by using the conversion functions **String To Path** and **Path To String**. There are also several functions in the File I/O function palette to assist you in this. Figure 7.1 shows how the **Build Path** function can append a string to an existing path. You might use this if you had a predefined directory where a file should be created and a file name determined by your program. Given a valid path, you can also parse off the last item in the path by using the **Strip Path**



**Figure 7.1** LabVIEW has functions to build, parse, and compare path names. These are useful when you are programmatically manipulating file names.

**Path** function. In Figure 7.1, the path is parsed until you get an empty path. Constants, such as **Empty Path** and **Not A Path**, are useful for evaluating the contents of a path name. The **Default Directory constant** leads you to a location in the file system specified through the LabVIEW preferences. You can also obtain a path name from the user by calling the **File Dialog** function. This function allows you to prompt the user and determine whether the chosen file exists.

Once you have selected a valid path name, you can use the **Open/Create/Replace File** function either to create a file or to gain access to an existing one. This function returns a **file refnum**, a magic number that LabVIEW uses internally to keep track of the file's status. This refnum, rather than the path name, is then passed to the other file I/O functions. When the file is finally closed, the refnum no longer has any meaning, and any further attempt at using it will result in an error.

You can navigate and directly manipulate your computer's file system from within LabVIEW. In the Advanced File palette are several useful functions, such as **File/Directory Info**, **List Folder**, **Copy**, **Move**, and **Delete**. This is a pretty comprehensive set of tools, although the methods by which you combine them and manipulate the data (which is mostly strings) can be quite complex.

## File Types

LabVIEW's file I/O functions can read and write virtually any file format. The three most common formats are

- ASCII text-format byte stream files
- Binary-format byte stream files
- LabVIEW datalog-format files

**ASCII text files** are readable in almost any application and are the closest thing to a universal interchange format available at this time. Your data must be formatted into strings before writing. The resulting file can be viewed with a word processor and printed, so it makes sense for report generation problems as well. A parsing process, as described in the strings section, must be used to recover data values after reading a text file. The disadvantages of text files are that all this conversion takes extra time and the files tend to be somewhat bulky when used to store numeric data.

**Binary-format byte stream files** typically contain a bit-for-bit image of the data that resides in your computer's memory. They cannot be viewed by word processors, nor can they be read by any program without detailed knowledge of the files' format. The advantage of a binary file is that little or no data conversion is required during read and write

operations, so you get maximum performance. Binary files are usually much smaller than text files. For instance, a 1000-value record of data from a data acquisition board may be stored in its raw 16-bit binary file, occupying 2000 bytes on disk. Converted to text, this record may take up to 10,000 bytes, or 5 times as much. The disadvantage of binary files is their lack of portability—always a serious concern.

LabVIEW offers another option, the **datalog-format file**, which is a special binary format. This format stores data as a sequence of records of a single arbitrary data type that you specify when you create the file.

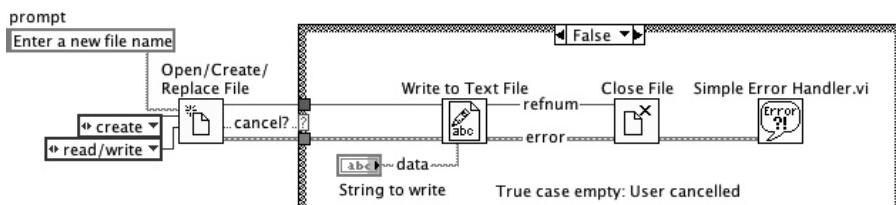
LabVIEW indexes data in a datalog file in terms of these records. Note that these records can be a complex type, such as a cluster, which contains many types of data. LabVIEW permits random read access to datalog files, and you can include timestamps with each record.

## Writing Text Files

Here's the basic procedure for most situations that write data to text files:

1. Determine the path name.
2. Open or create the file.
3. Convert the data to a string if it's not already in string format.
4. Write the string to the file.
5. Close the file.

Figure 7.2 uses several of the built-in file I/O functions to perform these steps. First, the user receives a dialog box requesting the name and location of a new file. The **Open/Create/Replace File** function has an **operation mode** input that restricts the possible selections to existing files, nonexisting files, or both. In this example, we wanted to create a new data file, so we set the mode to *create*, which forces the user to choose a new name. The function creates the desired file on the specified path, and returns a refnum for use by the rest of the file I/O functions. An output from the Open/Create/Replace File function,



**Figure 7.2** The built-in file I/O functions are used to create, open, and write text data to a file. The Case structure takes care of user cancellations in the file dialog.

**canceled**, is True if the user clicks the Cancel button. We wired it to a Case structure—its True case is empty—to skip the file I/O process if the operation was canceled. Next, the **Write To Text File** function writes the already formatted string. After writing, the **Close File** function flushes the data to disk and closes the file. Finally, we call **Simple Error Handler** (from the Dialog and User Interface function palette) to notify the user if an error occurs.

Note the clean appearance of this example, thanks to the use of the flow-through refnum and error I/O parameters. *Rule: Always include an error handler VI to do something sensible when an I/O error occurs.* Each of the file I/O functions returns an error code in its error I/O cluster. Not handling errors is a risky proposition for file I/O activity. Lots of things can go wrong. For instance, the disk might be full or a bad path could be generated if your path selection technique is complicated. *Rule: Always use error I/O (error chaining) for file I/O operations, when available.*

Your VIs don't have to be cluttered with lots of file management functions. A practical example that uses an open refnum to append text data to an open file is shown in Figure 7.3. Once a file has been opened and a refnum created, LabVIEW uses the open refnum to access the file's **Read Mark** and **Write Mark**. Inside the While Loop, a subVI reads data from some source and returns it in text format. Again, Write To Text File is called, and the data is appended at the Write Mark (the end of the file). The loop executes once per second, regulated by Wait Until Next ms Multiple. The result is a file containing a header followed by records of data at 1-s intervals.

This text-based scheme has some performance limitations. If you need to save data at really high rates (such as thousands of samples per second), the first thing to consider is binary-format files, which are very fast. If you must use text files, avoid closing the file after writing each record.

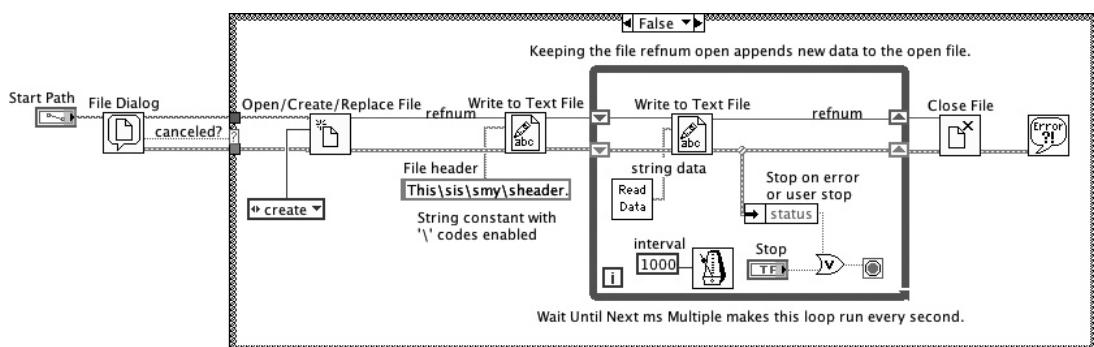
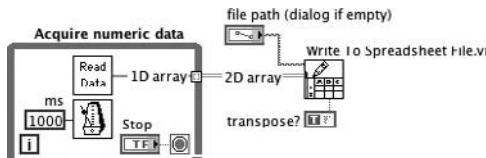


Figure 7.3 A simple data logger that uses an open refnum for all the file management. In the loop, data is acquired, formatted into a suitable string, and appended to the file. Simple, eh?



**Figure 7.4** Here's a simple way to log data to spreadsheet-compatible files by using a file utility.

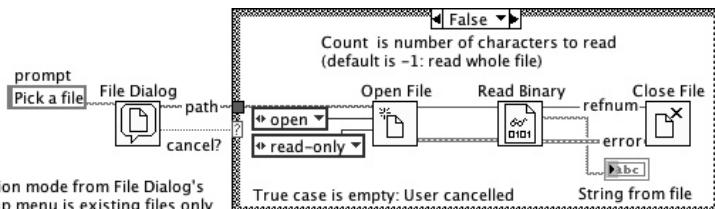
The file system maintains a memory-based file buffer, called a *disk cache*, which increases disk performance. Closing the file flushes the file buffer to disk. This forces the disk to move the heads around (called *seeking*) and then write the data, a time-consuming process that limits throughput. If you avoid closing the file until you are done, the disk cache can keep a moderate amount of data in memory, flushing to disk only when it becomes full. The disadvantage of using the disk cache is that if your computer crashes during data collection, data in the disk cache will be lost. For this reason, LabVIEW has a file I/O function **Flush File** that intentionally writes the output buffer to disk. You call **Flush File** occasionally to limit the amount of data in the buffer.

The file utility VI **Write To Spreadsheet File** can simplify your life when your objective is to write spreadsheet-compatible text. You often collect data in 1D or 2D numeric arrays, and you need to convert them to tab-delimited text. As shown in Figure 7.4, you can accumulate 1D arrays by indexing on the border of a loop and then wire it right into **Write To Spreadsheet File**. This is really convenient for exporting data from simulations that create 1D or 2D arrays. The **Transpose** option swaps rows and columns; it seems that you almost always need to transpose. One risk of using this VI as shown here is that LabVIEW may run out of memory if you let the While Loop run for a sufficiently long time. Instead, you can put **Write To Spreadsheet File** inside the acquisition loop: The VI will also accept 1D arrays, in which case it writes a single line of data values, separated by tabs and with a trailing end-of-line (EOL) marker. Another risk of using this VI is the time and additional memory required to convert from numeric to text. A 2-mega-sample array can quickly swell from 8 megabytes as SGL data to 32 megabytes or larger when converted to ASCII.

## Reading Text Files

Reading data from a text file is similar to writing it:

1. Determine the path name.
2. Open the file.
3. Read the string from the file.



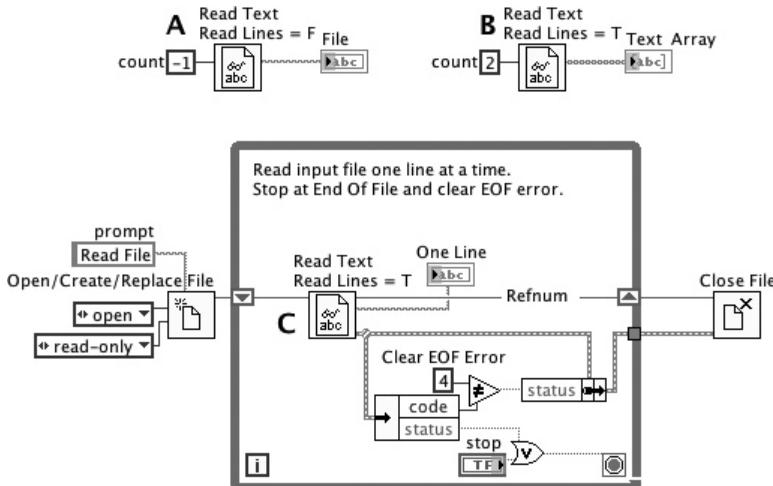
**Figure 7.5** Use the Read Binary function to read text file from disk.

4. Close the file.
5. Convert the string to a suitable format, if necessary.

Figure 7.5 shows an example that is symmetric with Figure 7.2. The **File Dialog** function provides a dialog box that only lets the user pick an existing file. This is configured from File Dialog's pop-up configuration menu which is accessible from the block diagram. Open File opens the file for read-only access and passes the refnum to Read Binary. The Read Binary function can read any number of bytes wired to the **count** input. By default, Read Binary starts reading at the beginning of the file, and **count** is  $-1$  (it reads all data). The default data type returned is string. For the simple case where you wish to read part of the file, just wire a constant to **count** and set it to the expected amount of data. We're using Read Binary in this example instead of **Read Text** because the default behavior of Read Text is to stop at the first EOL. Figure 7.6 shows several ways to use the **Read Text** function.

In the Read Text pop-up menu you select whether to read bytes or lines, and whether or not to convert EOL (Windows = CR/LF, Mac OS = CR, Linux = LF) characters. The default behavior is to convert EOL and read lines. Recognizing lines of text is a handy feature, but may not be what you expect when Read Text returns only the first line of your file. Figure 7.6A shows Read Text configured not to read lines of text. Setting count equal to  $-1$  reads an entire file. In Figure 7.6B, Read Text is configured to read lines and returns the number of lines equal to count as an array. In Figure 7.6C, Read Text increments through a text file one line at a time. This can be especially useful when you are incrementing through a text file containing an automated test sequence.

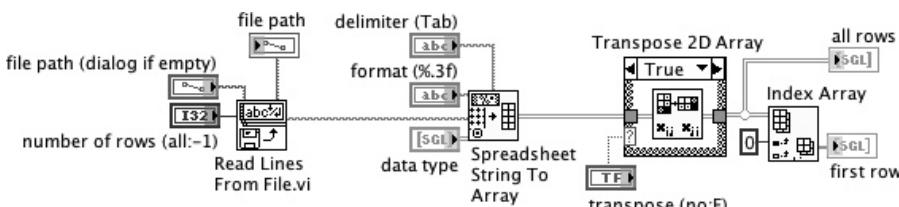
Another useful file utility VI is **Read From Spreadsheet File**, shown in simplified form in Figure 7.7. This VI loads one or more lines of text data from a file and interprets it as a 2D array of numbers. You can call it with **number of rows** set to 1 to read just one line of data at a time, in which case you can use the **first row** output (a 1D array). If the rows and columns need to be exchanged, set **transpose** to True. By default, it assumes that the tab character delimits values on each line.



**Figure 7.6** The Read Text function can read bytes or lines from a text file depending on its configuration. (A) It reads bytes ( $-1 = \text{all}$ ). (B) It reads lines. (C) It reads a file one line at a time.

If your format is different, you can enter the appropriate character in the **delimiter** string. Like many of the utilities, this one is based on another utility subVI, **Read Lines From File**, that understands the concept of a line of text. The end-of-line character (characters), which varies (vary) from system to system, is (are) properly handled. This functionality is now actually built into the Read Text function, as illustrated in Figure 7.6. The Read Lines From File goes on to read multiple lines until the end of the file is encountered.

*Words of wisdom:* Always remember that you can edit the diagram of a utility VI such as this to accommodate your special requirements. It's a powerful way to program because the groundwork is all done for you. But also remember to save the modified version with a new name, and save it someplace besides the vi.lib directory. If you don't, your next LabVIEW upgrade may overwrite your changes.



**Figure 7.7** The file utility VI **Read From Spreadsheet File.vi** interprets a text file containing rows and columns of numbers as a matrix (2D array). This is a simplified diagram.

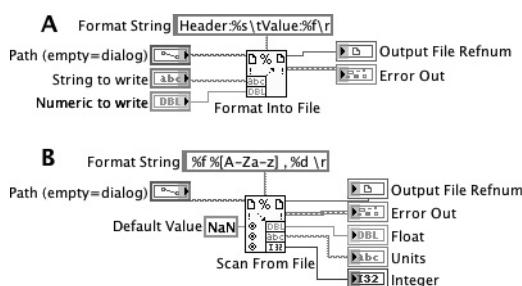
## Formatting to Text Files

There's another way to quickly read and write small amounts of text in files with control over the formatting. In Figure 7.8A, we used the **Format Into File** function from the Strings palette to write a formatted ASCII string into a new file. This function works like the Format Into String function in conjunction with file functions that write the data to disk; it's a great space saver. You supply a *Format String* that contains all the usual C-style format specifiers, and then you wire up as many input items as you like. It's limited to scalars, so you'll have to use other functions, such as Array To Spreadsheet String, if you need to write an array. This example shows a path control to choose the file, but you can also supply a file refnum if you already have a file open for writing.

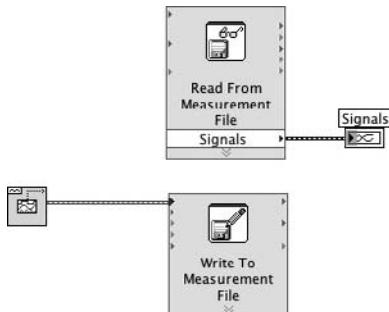
Figure 7.8B shows the complementary function, **Scan From File**, doing some on-the-fly parsing of a string as it is loaded from a text file. In this case, we expect a string that looks like this

12.34V,9

followed by a carriage return. It might represent a measured value (12.34) followed by a single character for the units (V), then a comma as a delimiter, and finally a scale factor (9). The Format String was created by popping up on the function and selecting Edit Scan String. That helps to guide you through the process of building the sometimes cryptic collection of format specifiers. The trickiest part here is the match for any single character in a set comprising all upper- and lowercase letters: %[A-Z,a-z]. Most of us mere mortals have trouble remembering such details, so we use the dialog. You can keep expanding Scan From File to handle an arbitrary number of items. Again, it doesn't scan arrays; you'll have to load those as strings and crack them with Spreadsheet String To Array, or use one of the other file utilities.



**Figure 7.8** The (A) Format Into File and (B) Scan From File functions are great for accessing files containing text items with simple formatting.



**Figure 7.9** Express VIs Read From Measurement File.vi and Write To Measurement File.vi take care of a lot of the work of reading and writing to files. They have a special LabVIEW format \*.lvm for text and \*.tdm for binary.

A recent addition to LabVIEW includes native measurement file formats for text and binary files. These new formats include the **metainformation** (who, what, when, how) about the measurement in the measurement file. The **Write To Measurement File.vi** and **Read From Measurement File.vi** convert to and from ASCII text files (\*.lvm) and binary files (\*.tdm). The text files include metainformation in a header within each file while the binary scheme uses an XML file to describe binary data stored in a separate data file. The express VI Write To Measurement File.vi includes a lot of functionality, such as the ability to automatically generate a new file name each time the VI is run. This makes it easy to acquire sequences of data in a loop and give each run a new file name. Figure 7.9 shows the Read From Measurement File and Write To Measurement File express VIs.

## Binary Files

The main reasons for using **binary files** as opposed to ASCII text are that (1) they are faster for both reading and writing operations and (2) they are generally smaller. They are faster because they are smaller and because no data conversion needs to be performed (see the section on conversions). Instead, an image of the data in memory is copied byte for byte out to the disk and then back in again when it is read. Converting to ASCII also requires more bytes to maintain numerical precision. For example, a single-precision floating-point number (SGL; 4 bytes) has 7 significant figures. Add to that the exponent field and some ± signs, and you need about 13 characters, plus a delimiter, to represent

it as ASCII. In binary you only need to store the 4 bytes—a savings of more than 3 to 1. Byte stream binary files (as opposed to datalog files) are also **randomly accessible**. That means you can tell the disk drive to read or write particular areas or individual bytes in the file. Random access is a bit tricky, however, as we shall see.

One small problem is that the program reading the data must know every little detail about how the file was written. Data types, byte counts, headers, and so forth have to be specified. You can't view it as an ASCII text string, so there's little hope of deciphering such a file without plenty of information. Even LabVIEW datalog files (a special type of binary file, discussed later) are not immune to this problem. If you can't tell the file I/O functions what data type the datalog file contains, it can't be accessed. On the other hand, consider the fact that parsing an arbitrary ASCII text file is also nontrivial, as we discussed in the section on strings. That's why there aren't any universal text file interpreter VIs.

You should consider using binary files when

- Real-time performance is crucial. Avoiding the numeric-to-text conversion and the extra amount of data that accompanies an ASCII text file will definitely speed things up.
- You already have a requirement for a certain binary format. For instance, you may be writing files in the native format of another application.
- Random read-write access is required.

To make a binary file readable, you have several options. First, you can plan to read the file back in LabVIEW because the person who wrote it should have darned well be able to read it. Within LabVIEW, the data could then be translated and written to another file format or just analyzed right there. Second, you can write the file in a format specified by another application. Third, you can work closely with the programmer on another application or use an application with the ability to import arbitrary binary formatted files. Applications such as Igor, S, IDL, MATLAB, and Diadem do a credible job of importing arbitrary binary files. They still require full information about the file format, however. These are several common data organization techniques we've seen used with binary files:

1. One file, with a header block at the start that contains indexing information, such as the number of channels, data offsets, and data lengths. This is used by many commercial graphing and analysis programs.

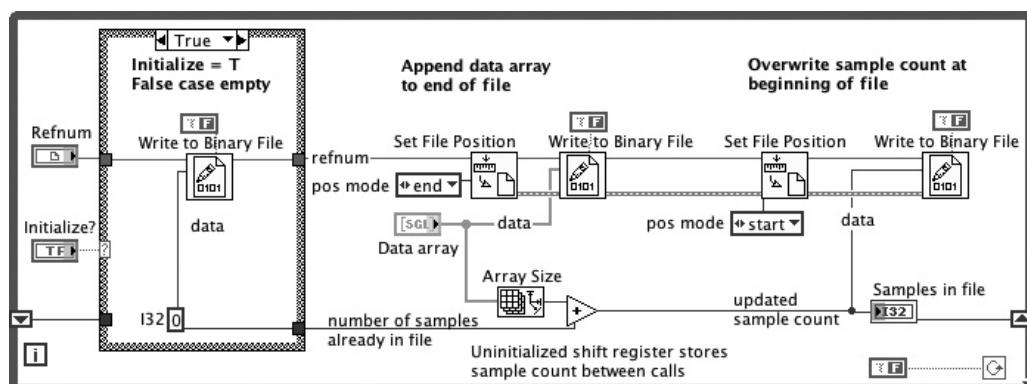
2. One file, organized as a *linked list*. Each record (say, a channel of data) is written with its own header that tells the reader both where this record begins and ends and where to find the next record in the file (this is called a *link*). This technique is used by some graphing and database programs.
3. Two files, one containing only the actual binary data and another containing an index to the data, perhaps in ASCII format. This is the format used by LabVIEW in \*.tdm files.

In all these formats, the index or header information is likely to include such things as channel names, calibration information, and other items you need to make sense of the data. Binary files are much more than a big bag of bytes.

### Writing binary files

LabVIEW's file I/O functions make it easy to access binary files. The effort, as with text files, is all in the data formatting. You have to figure out how to deal with your particular header or index format, and of course the data itself. Sorry, but that's *your* problem, since no two formats are the same. The best we can do is to show you the basics of random access files.

Figure 7.10 is a simple driver VI for a binary file format that we just made up. This format has a header containing one I32 integer (4 bytes) that is the number of data samples to follow. The data follows immediately afterward and is in SGL floating-point format. Each time this driver is called, the data array is appended to the file and the header is updated to reflect the total sample count.



**Figure 7.10** This subVI appends an array of SGL floats to a binary file and updates the sample count, an I32 integer, located at the beginning of the file. (Error handling is omitted for clarity.)

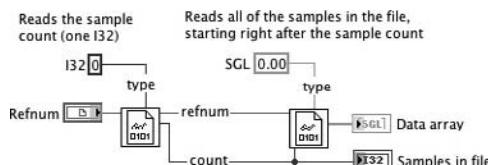
An important concept in file access is the **file mark**, which is a pointer to the current location in the file, measured in bytes. It is maintained invisibly by the file system. The **Set File Position function** permits you to place the mark anywhere you wish. This is how you achieve random access: Figure out where you are, then move to where you need to be. If the mark is at the end of the file (EOF) and you call Write To Binary File, then data is appended. If the mark is in the middle of the file, then Write To Binary File will overwrite existing data, starting at the mark.

In this example, the calling VI opens and closes the data file and passes the file's refnum via a refnum control. There are no special tricks to opening or creating a byte stream binary file; LabVIEW does not differentiate between binary and text files with regard to file type—it's only how you interpret the data that makes a difference. On the first call to this VI, **Initialize** should be set to True, which results in the sample count being set to 0 and written to the file. An uninitialized shift register stores the sample count for use on subsequent calls.

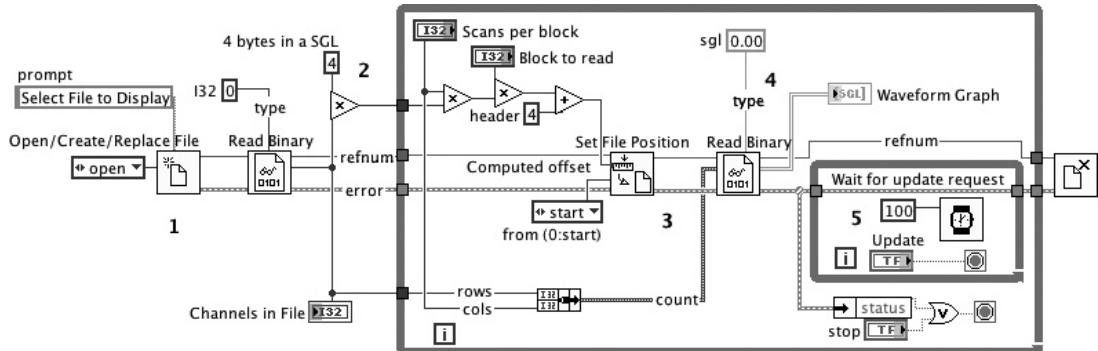
After initialization, Set File Position sets the write mark to the EOF, and Write To Binary File appends data to the end of the file. Since Write To Binary File is polymorphic, we just wired the incoming data array to the **data** input. The size of the array is added to the previous sample count. Then the Write To Binary File is called again, but this time Set File Position has set the write mark to the start of the file, which forces it to overwrite at the start of the file. The data in this case is our updated sample count, an I32 integer, which overwrites the previous value without bothering data elsewhere in the file.

## Reading binary files

Reading the data is even easier, at least for this simple example, as shown in Figure 7.11. Read File is called with the **type** input wired to an I32 constant so that it returns the sample count with the appropriate interpretation. This is another example of polymorphism at work.



**Figure 7.11** Reading the simple binary file created in the previous example. All you have to do is read the first 4 bytes, which represent the number of samples to follow, then read that number of data points. (Error handling is omitted for clarity.)



**Figure 7.12** This VI reads random blocks of data from a binary file written by the LabVIEW DAQ example VI, **Cont Acq to File (scaled)**. I couldn't make it much simpler than this.

If you do not wire to **type**, then the default type, *string*, is used, and your data is returned as a string. Next, Read File is called again to read the data, which is in SGL format. Because **count** is wired, Read File returns an array of the specified type. No special positioning of the file marker was required in this example because everything was in order. Things are more complex when you have multiple data arrays and/or headers in one file.

Random access reading is, as we said earlier, a bit tricky because you have to keep track of the file mark, and that's a function of the data in the file. To illustrate, we created a fairly simple VI that allows you to read random blocks of data from a file created by the LabVIEW DAQ example VI, **Cont Acq to File (scaled)**, located in the directory examples/daq/analogin/strmdisk.llb/. The binary file created by that example contains an I32 header whose value is the number of channels, followed by scans of SGL-format data. A scan consists of one value for each channel. The objective is to read and display several scans (we're calling this a *block* of data), starting at an arbitrary location in the file. Figure 7.12 is a solution (not the best, most feature-packed solution, we assure you, but it's simple enough to explain). Here's how it works.

1. The binary file is opened.
2. A single I32 integer is read by the Read Binary function. Its value is the number of channels in each scan. For computation of file offset, it is multiplied by 4, the number of bytes in a value of type SGL. Already, you can see the dependency on prior knowledge of the file's contents.
3. The program idles in a While Loop. On each iteration, a new file offset is computed. **Offset** equals scans per block, times the number of bytes per scan, times the desired block number, plus 4 bytes (which accounts for the I32 header). Read Binary is set up to read relative to the start of the file with the computed offset. We used another cool

feature of this file VI: It understands the concept of rows and columns when reading 2D arrays. You can see the **row** and **col** values on the diagram bundled into the **count** input of Read File.

4. The data is loaded and displayed on a waveform graph. You must pop up on the graph and select **Transpose Array**; otherwise, the *x* and *y* axes are swapped.
5. A small While Loop runs until the user clicks the Update button.

When the program is done, the file is closed and errors are checked. As you can see, even this simple example is rather tricky. That's the way it always is with binary files.

## Writing Datalog Files

LabVIEW datalog files conveniently store data in binary format one record at a time. The datalog file I/O functions are polymorphic, and record can be any LabVIEW data type, although you will usually use a cluster containing several data types. The datalog functions are available from the File I/O/Datalog palette. Datalog files require you to make a connection to the **type** input of Open/Create/Replace Datalog function. When the file is opened, the refnum (properly called a **datalog refnum**) carries information that describes the type of data in the file. The **Read Datalog** and **Write Datalog** functions then adapt to the appropriate type. A broken wire will result if you attempt to wire the **data** terminal to a source or destination of the wrong type.

Figure 7.13 is a rework of our simple data logger using LabVIEW datalog files. At the lower left, the data type is specified as a cluster

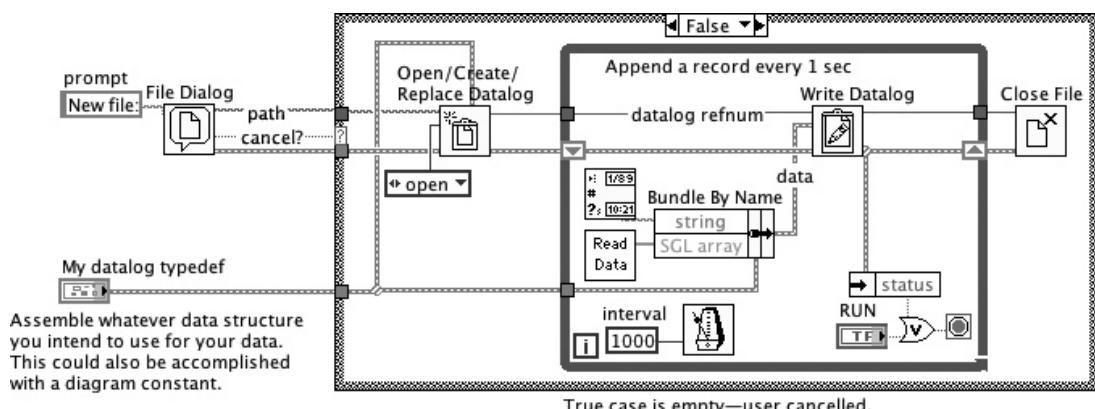


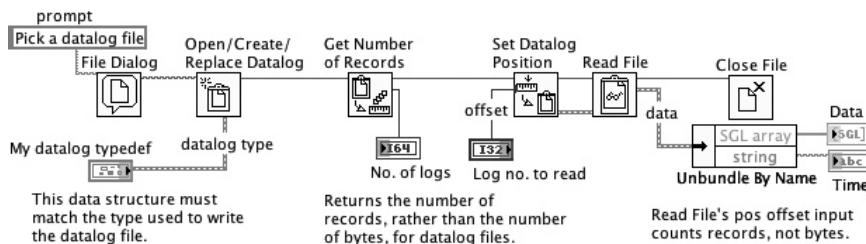
Figure 7.13 Writing data to a datalog file. The data type in this case is a cluster containing a string and a numeric array. Records are appended every second. (Error handling is removed for clarity.)

containing a string and an SGL numeric array. **Open/Create/Replace** Datalog responds by creating a file with a datalog refnum that corresponds to this type. In the While Loop, Write Datalog appends one record per iteration, containing a string with the current time and an array of values. The datalog refnum tells Write Datalog what data type to expect. Note that datalog files are effectively *append only*. To edit the contents of a datalog file, you must open the file and copy the desired records to another, similar datalog file.

Note the use of a **typedef** control for determination of data type in this example. By wiring such a type specifier to all dependent objects (Open/Create/Replace Datalog, Write Datalog, and Bundle By Name), you can guarantee that the program will work properly with no broken wires, even if you change the constituents of the control. This is extremely important for code maintenance in large projects and is a good habit to get into. For this simple example, you wouldn't have to edit or save the control as a typedef, but at least it's properly wired and easy to change if necessary.

You can also create datalog files by using **front panel datalogging** functions under the Operate menu. Front panel datalogging appends a record containing a timestamp and all the front panel data to a file of your choosing. In the Datalogging submenu, you create a file (**Change Log File Binding ...**) and then manually log the panel at any time (the **Log ...** command) or enable **Log At Completion** to automatically append a record when execution of the VI ends. The VI need not have its front panel displayed for Log At Completion to function. This is a handy way to do datalogging without programming in the usual sense. Just create a subVI with a set of controls you wish to log (it does not need any wiring on its diagram). Set it up to Log At Completion, wire it into the diagram of a calling VI, and every time the subVI is called, a data snapshot is recorded. The format of this kind of datalog is a cluster containing two things: a timestamp and a cluster with all front panel controls. The cluster order is the same as the Tabbing Order, which you set in the **Edit >> Set Tabbing Order ...** menu item.

Front panel datalogging creates standard datalog files, readable by the methods described in the following section. You can also view the contents of each record right on the panel by choosing **Retrieve ...** from the Datalogging submenu. This is a convenient way of storing and manually retrieving VI setups. You can also place the VI from which you logged data onto a diagram, pop up on it, and select **Enable Database Access**. A halo that looks like a file cabinet appears around the icon of the subVI. It has terminals for programmatically accessing specified records and for reading timestamps and data.



**Figure 7.14** This VI reads a single datalog of the format written by the previous example. Note the value of creating a typedef control to set the datalog type. (Error handling is removed for clarity.)

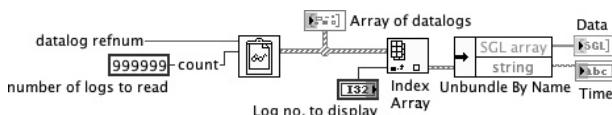
## Reading Datalog Files

You can read one or more records at a time from a datalog file. Figure 7.14 shows the basic method by which you read a single record. The **EOF** function reports the number of records in the file, rather than the number of bytes, as it did with a byte stream text file. Similarly, Read File responds to its **pos offset** input by seeking the desired record number (starting with zero) rather than a byte number. In this example, one record containing the expected bundle of data is returned. Note how Read File's data output is of the same type as the datalog typedef wired to Open File. The type information is carried in the datalog refnum.

If you want to read more than one record at a time, just connect a numeric value to the **count** terminal of Read File, as shown in Figure 7.15. Its output will then be an array of the data type previously defined. You can also wire the **pos offset** input to select which record is the first to be read. With these two controls, you can read any number of records (in sequence) from any location in the file.

## Datalog file utilities

An easy way to get started with datalog files is to use the utility VIs in the LabVIEW examples (path: labview/examples/file/datalog.llb). These VIs are modeled after some of the other file utilities and include functions to open, read, write, and close files containing datalogs with a data type of your choice. The examples simulate common usage, and there is a handy example that reads one record at a time when you click a button.



**Figure 7.15** This code fragment reads all the records from a datalog file.

*This page intentionally left blank*

---

Chapter  
**8**

## Building an Application

Many applications we've looked at over the years seemed as though they had just happened. Be it Fortran, Pascal, C, or LabVIEW, it was as if the programmer had no real goals in mind—no sense of mission or understanding of the big picture. There was no planning. Gary actually had to use an old data acquisition system that started life as a hardware test program. One of the technicians sat down to test some new I/O interface hardware one day, so he wrote a simple program to collect measurements. Someone saw what was going on and asked him if he could store the data on disk. He hammered away for a few days, and, by golly, it worked. The same scenario was repeated over and over for several years, culminating in a full-featured ... *mess*. It collected data, but nobody could understand the program, let alone modify or maintain it. By default, this contrivance became a standard data acquisition system for a whole bunch of small labs. It took years to finally replace it, mainly because it was a daunting task. But LabVIEW made it easy. These days, even though we start by writing programs in LabVIEW, the same old scenario keeps repeating itself: Hack something together, then try to fix it up, and document it sometime later.

Haphazard programming need not be the rule. Computer scientists have come up with an arsenal of program design, analysis, and quality management techniques over the years, all based on common sense and all applicable to LabVIEW. If you happen to be familiar with such formalisms as structured design, by all means use those methods. But you don't need to be a computer scientist to design a quality application. Rather, you need to think ahead, analyze the problem, and generally be methodical. This chapter should help you see the big picture

and design a better application. Here are the steps we use to build a LabVIEW application:

1. Define and understand the problem.
2. Specify the type of I/O hardware you will need.
3. Prototype the user interface.
4. Design—then write—the program.
5. Test and debug the program.
6. Write the documentation (*please!*).

To make this whole process clearer, we're going to go through a real example: a LabVIEW-based control and datalogging system Gary developed for Larry's Vacuum Brazing Lab (call it **VBL**, for short). At each step, we outline the general approach and then illustrate the particular tactics and the problems discovered in the VBL project. At the end of the chapter we look at the LabVIEW Certification exams.

## Define the Problem

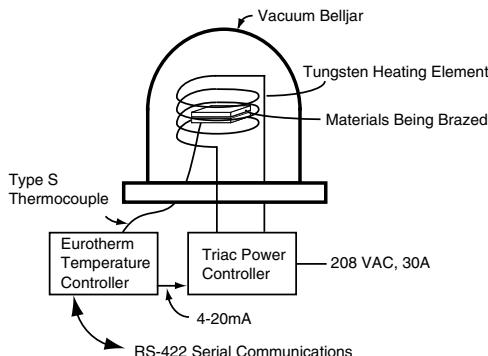
If you, the system designer, can't understand the problem, all is lost! You will stumble about in the dark, hoping to accidentally solve your customer's problems through blind luck or divine intervention.

Sorry, but that's not going to happen. What you need to do is to spend a significant amount of time just understanding the problem. At this point, you don't need a computer at all, just a pencil, maybe a voltmeter or an oscilloscope, and the desire to find out what is really going on.

## Analyze the user's needs

Interview the end users in several sessions. Talk to everyone involved, paying particular attention to the operators—the ones who actually have to interact with the VIs you write. Even if *you* are the customer, you still have to go through this step to make sure that you have all the important specifications on paper. How else will you know when you're done?

Tell your customers about LabVIEW. Show them what it can do in a live demonstration. That way, you can speak a common language—you can speak of controls, indicators, files, and subVIs. In return, your customer should show you the system that needs to be monitored or controlled. Perhaps there's an existing measurement and control system that has shortcomings—find out what's wrong with that existing system. Learn the proper terminology. Maybe you need to read up on the technology if it's something you've never worked with before.



**Figure 8.1** A simplified vacuum brazing furnace. The temperature controller adjusts power to the heating element until the measured temperature matches the desired set point. Not shown are the vacuum pump and other support equipment. Serial communications make it possible for LabVIEW to run this furnace.

Gary didn't know what vacuum brazing was until he spent a couple of hours with Larry. Here's what he learned:

The principle of vacuum brazing is simple (Figure 8.1). *Brazing* involves joining two materials (usually, but not always, metal) by melting a filler metal (such as brass) and allowing it to wet the materials to be joined. Soldering (like we do in electronics) is similar. The idea is *not* to melt the materials being joined, and that means carefully controlling the temperature.

Another complication is that the base metals sometimes react with the air or other contaminants at high temperatures, forming an impervious oxide layer that inhibits joining. Larry's solution is to put everything in a vacuum chamber where there is no oxygen, or anything else for that matter. Thus, we call it *vacuum brazing*. The VBL has five similar electric furnaces, each with its own vacuum bell jar. The heater is controlled by a Eurotherm model 847 digital controller, which measures the temperature with a thermocouple and adjusts the power applied to the big electric heater until the measured temperature matches the set point that Larry has entered.

### Gather specifications

As you interview the customers, start making a list of basic **functional requirements and specifications**. Watch out for the old standoff where the user keeps asking what your proposed system can provide.

You don't even *have* a proposed system yet! Just concentrate on getting the real needs on paper. You can negotiate practical limitations later.

Ask probing questions that take into consideration his or her actual understanding of the problem, his or her knowledge of instrumentation, and LabVIEW's capabilities. If there is an existing computer system (not necessarily in the same lab, but one with a similar purpose), use that as a starting point. As the project progresses, keep referring to this original list to make sure that everything has been addressed. (It's amazing, but every time I think I'm finished, I check the requirements document and find out I've missed at least one important feature.)

Review the specifications with the user when you think you understand the problem. In formal software engineering, the requirements document may adhere to some standards, such as those published by IEEE, which can be quite involved. In that case, you will have formal design reviews where the customer and perhaps some outside experts will review your proposed system design. Although it is beyond the scope of this chapter, here are some important requirements to consider:

- Characterize each signal. Is it digital or analog? Low or high frequency? What kind of I/O hardware is proposed or required for proper interfacing? Are LabVIEW drivers available?
- What are the basic operations, steps, event sequences, and procedures that the software needs to perform? This determines much of the overall structure of the program.
- Are there critical performance specifications? For example, rapid response time may be required in a control loop, or very high accuracy and noise rejection are desired. Make sure that the hardware, LabVIEW, and your programming expertise are up to the task.
- What displays and controls are required? Break down displays by type, such as charts, graphs, numerics, strings, and booleans. Consider which screen object controls or displays which signals.
- Determine the signal analysis needs such as statistics, linearization, peak detection, frequency transforms, filtering and so on. If unusual algorithms are required, seek details early in the project. What needs to be done in real time versus in postrun analysis? This may drastically affect your choice of computer and the system's overall performance.
- What will be done with the data that will be collected? Compatibility with other applications (sometimes on other computers) is an important issue. You must always save the data in a suitable format, and you should be involved with the development of analysis tools. Various options for real-time exchange with other programs may be

considered as well (dynamic data exchange; interapplication communication; networking).

- What other information needs to be recorded besides acquired data? Often, users like to save run setup information or comments in a running log format with timestamps.
- How much of this is *really* necessary? Separate needs from wants. Prioritize the tasks. Realize that there are *always* constraints on budget and/or time, and you have to work within these constraints. Speaking of which, get commitment on the budget and schedule early in the project!

The major requirements for the VBL project I divided into *must-haves* and *future additions*, which prioritized the jobs nicely. Larry was really short of funding on this job, so the future additions are low priority.

### *Must-Haves*

1. All five systems may operate simultaneously.
2. Procedure: ramp up to a soak setpoint, then go to manual control so user can tweak temperature up to melting point. Resume ramp on command.
3. LabVIEW will generate the temperature ramp profiles. User needs a nice way to enter parameters and maintain recipes.
4. Strip-chart indicators for real-time temperature trends.
5. Notify user (beep and/or dialog) when the soak temp is reached.
6. Alarm (beep and dialog) when temperature exceeds upper limit in controller.
7. Make post-run trend plots to paste into Microsoft Word report document.

### *Future Additions*

1. Control vacuum pump-down controller. This will require many RS-232 ports.
2. Trend vacuum measurement. This will require even more RS-232 ports.

### Draw a block diagram

Draw a preliminary **block diagram** of the system. Think in terms of signal flow all the way from I/O hardware to computer screen and back out again. What are the inputs and outputs? This will define the signals, and the signals will define the kind of I/O hardware and data presentation that you will need. List the characteristics of each signal.

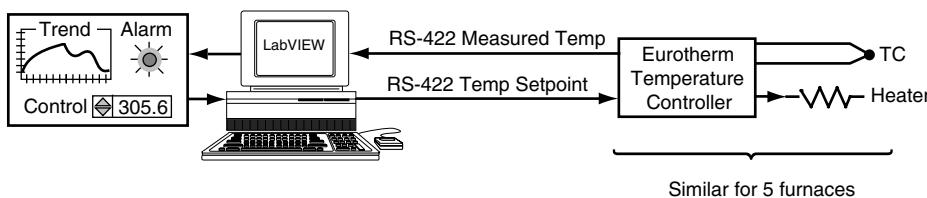
Some of the items to consider are as follows:

- Sensor and actuator types, manufacturers, model numbers, and so forth
- Number of channels, categorized by signal type
- Signal characteristics: voltage, current, pulse rate, and so forth
- Frequency content (determines sampling rates and filtering requirements)
- Isolation and grounding requirements
- Type of analysis to be performed (influences acquisition and storage techniques)

Get as much detail as possible right now. This information is vital if you hope to design a really successful automation package. It would be a shame to misspecify I/O hardware or write a program that simply didn't account for the needs of all the instruments you plan to support.

For VBL, I learned that there would only be two basic signals running between the computer and each furnace: the measured temperature from the controller and the setpoint going to the controller (Figure 8.2). In their raw forms, both are DC analog signals. The Eurotherm controller converts the signals to and from digital commands that are transmitted over an RS-422 serial communications line. RS-422 permits *multidrop connections*, meaning that multiple controllers can share a single communications line. Therefore, I knew that no signal conditioning of the usual type was needed, but a driver for this particular Eurotherm controller would have to be written. As a bonus, the communications hardware uses differential connections to enhance noise rejection—very thoughtful, these engineers.

Collect details on any special equipment proposed for use in the system. Technical manuals are needed to figure out how to hook up the signal lines and will also give you signal specifications. For all but the simplest instruments there may be a significant driver development



**Figure 8.2** Draw a block diagram of your project to make sure you know how the pieces fit together. This one was really simple, with only one input and one output for each of five furnaces.

effort required (especially for GPIB or serial communications), so you will need to get a copy of the programming manual as well. Sometimes one major instrument *is* the I/O system.

When you have all the information together, write up a system requirements document—even if it’s only one page. Even if your company does not require it. Even if it seems like too much work at the moment. Why? *Because it will pay off in the form of a higher-quality product.*

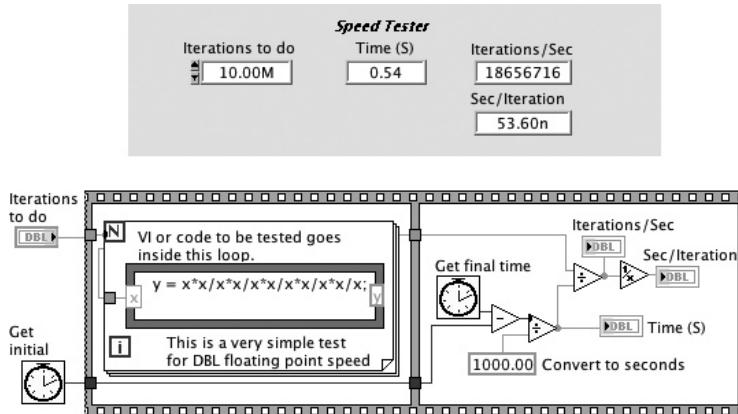
## Specify the I/O Hardware

Now that you know which instruments will be used on your project, follow the steps outlined in Chapter 12, “Inputs and Outputs,” to determine what kind of I/O hardware to use. The signals your instruments produce will define the hardware. There are many factors to consider, so choose wisely.

Since you have to write a LabVIEW program to support the I/O, having examples of completed **drivers** available will save you much time and effort. The LabVIEW instrument driver network ([www.ni.com/idnet](http://www.ni.com/idnet)) contains hundreds of ready-to-use drivers, all available for free. Even if your instrument isn’t there, there may be something very close—a different model by the same manufacturer—that is easily modified. If you plan to use any of the National Instruments (NI) multifunction boards, prowl through the DAQ example VIs to find relevant demonstration programs that you can use as starting points. For simple projects, it’s surprising how quickly the examples can be modified to become the final application.

Availability and performance of drivers are one more consideration in your choice of I/O hardware. You may need to do some preliminary testing with the actual instruments and data acquisition equipment before you commit to a final list of specifications. Be absolutely sure that you can meet the user’s needs for overall **throughput**: the rate at which data can be acquired, processed, displayed, and stored. Write some simple test VIs at this point and rest better at night. The **Speed Tester** VI in Figure 8.3 is useful when you have a subVI that you want to exercise at top speed. You can put anything inside the For Loop and get a fairly accurate measure of its execution time. Make sure that the speed tester does plenty of iterations so that you get enough timing resolution.

In most cases you can’t write the application unless you know what hardware will be used. For instance, in a high-throughput data acquisition application in which many thousands of samples per second of data are streamed to disk, you probably can’t do much on-the-fly analysis, graphics, or changing of windows because of the extreme performance



**Figure 8.3** A general-purpose speed tester VI that is used when you need to check the throughput or iteration rate of a subVI or segment of code. This example does 10 floating-point operations.

demands on the CPU. On the other hand, programming of a low-speed temperature recorder has no such demands and permits a substantial amount of real-time processing and display.

The Eurotherm controllers use the serial port for communications. Even though Larry was only asking for one sample per second top speed, I suspected that there could be an upper limit to the total throughput lurking close by. Serial instruments are notorious for adding lots of header and trailer information to otherwise simple messages, which compounds the problem of an already-slow communications link. The bad news was that there was no driver available, so I had to write one from scratch before any testing was possible. The good news was that it only took 30 ms to exchange a message with a controller. That means I could move about 33 messages per second, distributed among the five controllers, or about 6 messages per controller per second. Not a problem, unless he ever wants to triple the number of furnaces.

### Prototype the User Interface

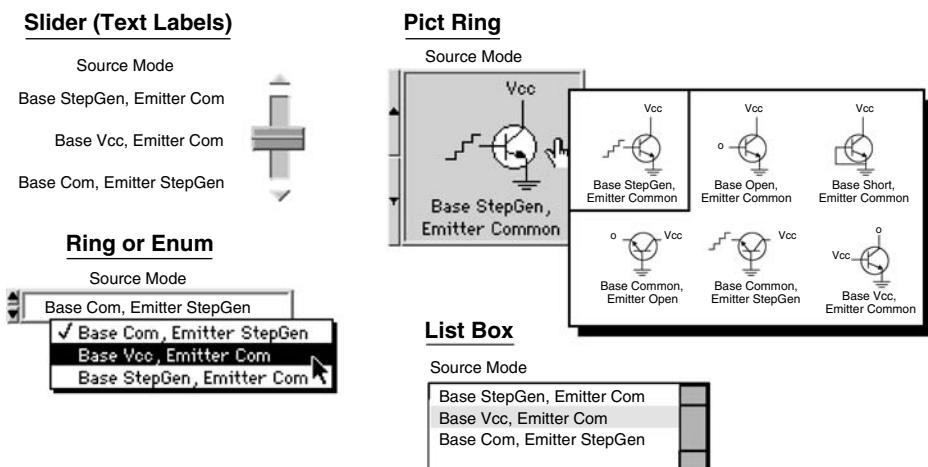
An effective and fun way to start designing your application is to create prototype front panels that you can show to the users. The panels don't have to work at all, or maybe you can just hook up random number generators to animate the graphs and other indicators. Don't spend much time programming; concentrate on designing panels that meet the specifications. Consider the best types of controls and indicators for each signal. Make a list of inputs and outputs, grouped in some logical categories, and pair them up with the most sensible control and indicator types. If you have to write a formal software requirements

document, you can paste images of your mock front panels into the document. That makes your intentions clearer.

### Panel possibilities

Remember that you have an arsenal of graphical controls available; don't just use simple numerics where a slider or ring control would be more intuitive. Import pictures from a drawing application and paste them into **Pict Ring** controls or as states in boolean controls, and so forth, as shown in Figure 8.4. Color is fully supported (except in this book). Deal with indicators the same way. Use the **Control Editor** to do detailed customization. The Control Editor is accessed by first selecting a control on the panel and then choosing **Customize Control** from the Edit menu. You can first resize and color all the parts of any control and paste in pictures for any part and then save the control under a chosen name for later reuse. Be creative, but always try to choose the appropriate graphic design, not just the one with the most glitz. After all, the operator needs information, not entertainment.

Booleans with pictures pasted in are especially useful as indicators. For instance, if the True state contains a warning message and the False state has both the foreground and background color set to transparent (**T**, an invisible pen pattern in the color palette when you are using the coloring tool), then the warning message will appear as if by magic when the state is set to True. Similarly, an item can appear to



**Figure 8.4** Controls can be highly customized with various text styles and imported pictures. Here are four sensible numeric controls that could be used for the same function. Pick your favorite.

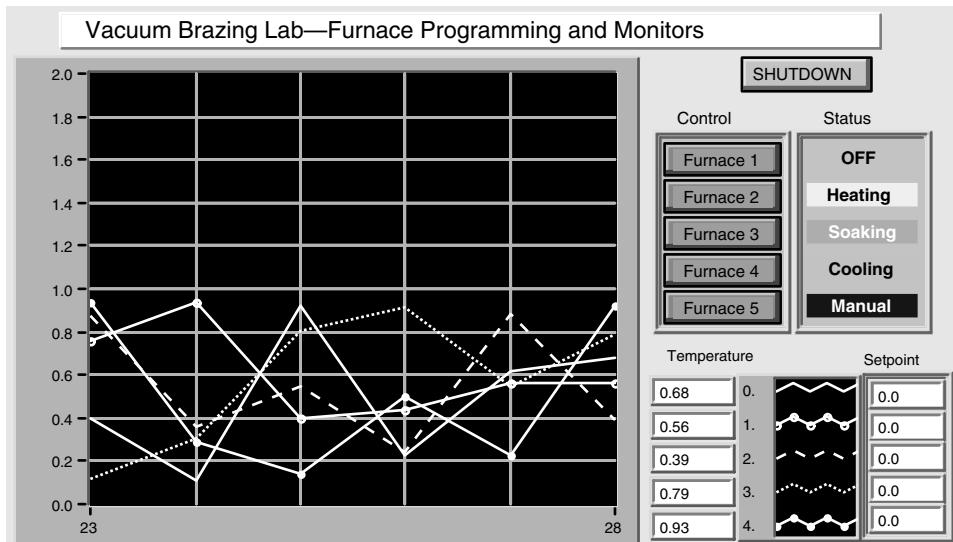
move, change color, or change size based on its state through the use of these pasted-in pictures. The built-in LabVIEW controls and indicators can be made to move or change the size programmatically through the use of **Property nodes**.

Real-time trending is best handled by the various charts, which are designed to update displayed data one point or one buffer at a time. Arrays of data acquired periodically are best displayed on a graph, which repaints its entire display when updated. Note that graphs and charts have many features that you can customize, such as axis labels, grids, and colors. If you have a special data display problem that is beyond the capability of an ordinary graph or chart, also consider the **Picture Controls**. They allow you to draw just about anything in a rectangular window placed on the front panel. There are other, more advanced data display toolkits available from third parties. Chapter 20, “Data Visualization, Imaging, and Sound,” discusses aspects of graphical display in detail.

Simple applications may be handled with a single panel, but more complex situations call for multiple VIs to avoid overcrowding the screen. Subdivide the controls and indicators according to the function or mode of operation. See if there is a way to group them such that the user can press a button that activates a subVI that is set to **Show front panel when called** (part of the VI Properties menu, available by popping up on the icon). The subVI opens and presents the user with some logically related information and controls, and it stays around until an Exit button is pressed, after which its window closes—just as a dialog box does, but it is much more versatile. Look at the Window options in the VI Properties menu. It contains many useful options for customizing the look of the VI window. Functions for configuration management, special control modes, alarm reporting, and alternative data presentations are all candidates for these dynamic windows. Examples of dynamic windows and programming tricks illustrating their use are discussed in Chapter 18, “Process Control Applications.” For demonstration purposes, these dynamic windows can be quickly prototyped and made to operate in a realistic manner.

Your next step is to show your mock-up panels to the users and collect their comments. If there are only one or two people who need to see the demonstration, just gather them around your computer and do it live. Animation really helps the users visualize the end result. Also, you can edit the panels while they watch (see Figure 8.5).

If the audience is bigger than your office can handle, try using a projection screen for your computer down in the conference room. LCD panels and red-green-blue (RGB) projectors are widely available and offer reasonable quality. That way, you can still do the live demonstration. As a backup, take screen shots of the LabVIEW panels that you



**Figure 8.5** Can you make that graph a little taller? “No sweat,” I replied, while calmly resizing the graph.

can edit in a draw/paint application and add descriptive notes. Besides making a nice handout, hard copies are especially useful for projects that require quality assurance plans and formal design reviews. You’ve got to have things in print, you know.

The Vacuum Brazing Lab had only a few things to display during normal operation: the five furnace temperatures, their setpoints, and their status (heating, soaking, cooling, etc.). That made one nice front panel with a big strip chart and a few other indicators. Next, I needed a panel through which Larry could edit the ramp-and-soak temperature recipe for each furnace—an ideal application for a dynamic window. Finally, there had to be a panel for manual control of the furnace. That panel needed such controls as auto/manual, recipe start/stop, and manual setpoint entry. It, too, was to be a dynamic window that appeared whenever a button on the main panel was pressed. I created rough versions of the three panels, made some dummy connections that animated it all, and showed it to Larry.

Iterate on your preliminary panel design. Big projects may require several demonstrations to satisfy the more critical users. This is especially true if you have collected additional requirements because of the impact of your presentation. People start seeing alternatives they previously had not imagined. Take notes during the demonstrations and add to your requirements list. Discuss the new ideas and requirements with your cohorts, and try to come up with likely programming

solutions to the more difficult problems. Remember to keep the users in the loop during all phases of the review process. They will be the ultimate judges of your work, and you certainly don't want to surprise them on the day of final delivery.

## First Design and Then Write Your Program

Don't be stymied by the "blank screen syndrome." The same effect occurs in many fields when it's time to create something: the writer with a blank sheet of paper or the painter with an empty canvas. Prototyping the user interface takes care of the panel, but the diagram starts as a big collection of unconnected controls and indicators. Where should you start?

Modern software engineering starts a project in the same way as any other engineering discipline: with a **design**—a plan of attack. You must do the same. The alternative is to sit in front of your computer, hoping for divine inspiration to somehow guide you toward the optimum program. Most likely, you'll end up with another of those little test programs that balloon into full-featured, unmaintainable disasters.

Instead, let's try using some design practices that work well in the LabVIEW environment. See *LabVIEW Power Programming* (Johnson 1998) for detailed discussions of software engineering in LabVIEW.

In case you're wondering why designing and writing the program are wrapped into this one section instead of being separated, there is a good reason: Most LabVIEW VIs run nicely on their own, with no main program. This is an important LabVIEW advantage that is often overlooked. Once you get past the most general stages of design, you start writing subVIs that beg to be run, tested, and optimized as you go. There is no need to write elaborate test programs, no need to wait until the entire application is assembled, complete with a bunch of buried bugs.

The ability to do **rapid prototyping** in LabVIEW is one of the major reasons for the product's success. You can quickly throw together a test program that improves your understanding of the underlying hardware and the ways in which LabVIEW behaves in your particular situation. These prototype programs are extremely valuable in the overall development process because they allow you to test important hypotheses and assumptions before committing yourself to a certain approach. Your first try at solving a problem may be ugly, but functional; it's the *later iterations* of this initial program that you actually deliver. This concept of **iterative design** is often the only practical approach to solving laboratory problems where the requirements change as fast as you can modify the program. If you can't seem to arrive at a concrete list of requirements, just do the best you can to design a flexible

application that you can update and extend on short notice. Many of the architectures shown in Chapter 3, “Controlling Program Flow,” are based on such practical needs and are indeed flexible and extensible.

If you’re an experienced programmer, be sure to use your hard-learned experience. After all, LabVIEW is a programming language. All the sophisticated tools of structured analysis can be applied (if you’re comfortable with that sort of thing), or you can just draw pictures and block diagrams until the functional requirements seem to match. This world is too full of overly rigid people and theorists producing more heat than light; let’s get the job done!

### Ask a Wizard

To help you get started building data acquisition applications, National Instruments added a **DAQ Assistant** to LabVIEW. The Wizard leads you through a list of questions regarding your hardware and application in the same way you might interview your customer. Then it places a DAQ solution that is ready to run. From that point, you continue by customizing the user interface and adding special features. As a design assistant, it’s particularly valuable for DAQ-based applications, but it can also suggest basic program designs that are useful for ordinary applications. As time goes on, LabVIEW Wizards will expand to encompass a broader range of programs.

### Top-down or bottom-up?

There are two classic ways to attack a programming problem: from the top down and from the bottom up. And everyone has an opinion—an opinion that may well change from one project to the next—on which way is better.

**Top-down** structured design begins with the big picture: “I’m going to control this airplane; I’ll start with the cockpit layout.” When you have a big project to tackle, top-down works most naturally. LabVIEW has a big advantage over other languages when it comes to top-down design: It’s easy to start with the final user interface and then animate it. A top-down LabVIEW design implies the creation of dummy subVIs, each with a definite purpose and interrelationship to adjacent subVIs, callers, and callees, but at first without any programming. You define the front panel objects, the data types, and the connector pane and its terminal layout. When the hierarchy is complete, then you start filling in the code. This method offers great flexibility when you have teams of programmers because you tend to break the project into parts (VIs) that you can assign to different people.

**Bottom-up** structured design begins by solving those difficult low-level bit manipulation, number-crunching, and timing problems right

from the start. Writing an instrument driver tends to be this way. You can't do anything until you know how to pass messages back and forth to the instrument, and that implies programming at the lowest level as step 1. Each of these lower-level subVIs can be written in complete and final form and tested as a stand-alone program. Your only other concern is that the right kind of inputs and outputs be available to link with the calling VIs.

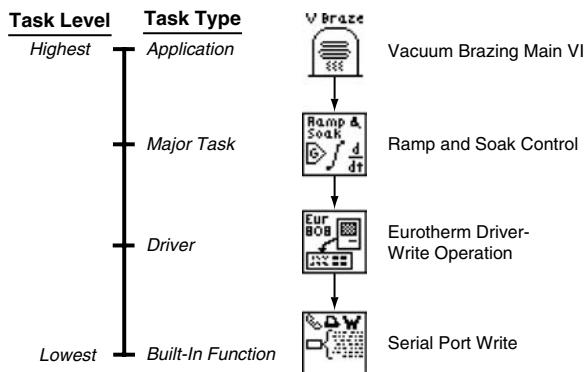
Keeping track of inputs and outputs implies the creation of a **data dictionary**, a hierarchical table in which you list important data types that need to be passed among various subVIs. Data dictionaries are mandatory when you are using formal software design methods, but are quite useful on any project that involves more than a few VIs. It is easy to maintain a data dictionary as a word processing document that is always open in the background while you are doing the development. A spreadsheet or database might be even better; if you use a computer-aided software engineering (CASE) tool, it will of course include a data dictionary utility. List the controls and indicators by name, thus keeping their names consistent throughout the VI hierarchy. You might even paste in screen shots of the actual front panel items as a reminder of what they contain. This is also a chance to get a head start on documentation, which is always easier to write at the moment you're involved with the nuts and bolts of the problem. Many items that make it to the dictionary are saved as **typedefs** or **strict typedefs** that you define in the Control Editor. This saves countless hours of editing when, for instance, a cluster needs just one more element.

Don't be afraid to use both top-down and bottom-up techniques at once, thus ending up in the middle, if that feels right. Indeed, you need to know *something* about the I/O hardware and how the drivers work before you can possibly link the raw data to the final data display.

## Modularity

Break the problem into modular pieces that you can understand. This is the divide-and-conquer technique. The modules in this case are **subVIs**. Each subVI handles a specific **task**—a function or operation that needs to be performed. Link all the tasks together, and an **application** is born, as shown in Figure 8.6.

One of the tricks lies in knowing *when* to create a subVI. Don't just lasso a big chunk of a diagram and stick it in a subVI because you ran out of space; that only proves a lack of forethought. Wouldn't you know, LabVIEW makes it easy to do: select part of your diagram, choose **Create SubVI** from the Edit menu, and poof! Instant subVI, with labeled controls and connector pane, all wired into place. Regardless of how



**Figure 8.6** Using modularity in your program design makes it much easier to understand. This is one slice through the VI hierarchy of the VBL project.

easy this may be, you should instead always think in terms of tasks. Design and develop each task as if it were a stand-alone application. That makes it easier to test and promotes reuse in other problems. Each task, in turn, is made up of smaller tasks—the essence of top-down hierarchical design.

Any properly designed task has a clear purpose. Think of a one-sentence thesis statement that clearly summarizes the purpose of the subVI: “This VI loads data from a series of transient recorders and places the data in an output array.” (A good place to put this thesis statement is in the VI Documentation dialog.) If you can’t write a simple statement like that, you may be creating a catchall subVI. Include the inputs and outputs and the desired behavior in your thesis statement, and you have a VI specification. This enables a group of programmers to work on a project while allowing each programmer to write to the specification using her or his own style. As an example of how powerful this is, a recent coding challenge posted to the LabVIEW Developer Zone ([www.ni.com](http://www.ni.com)) had more than 100 submissions; although each submission was different, they all did essentially the same thing. More than 100 programmers, and all their code was interchangeable because they all wrote to the same specification. Not only does it make team development possible, but also maintenance is easier.

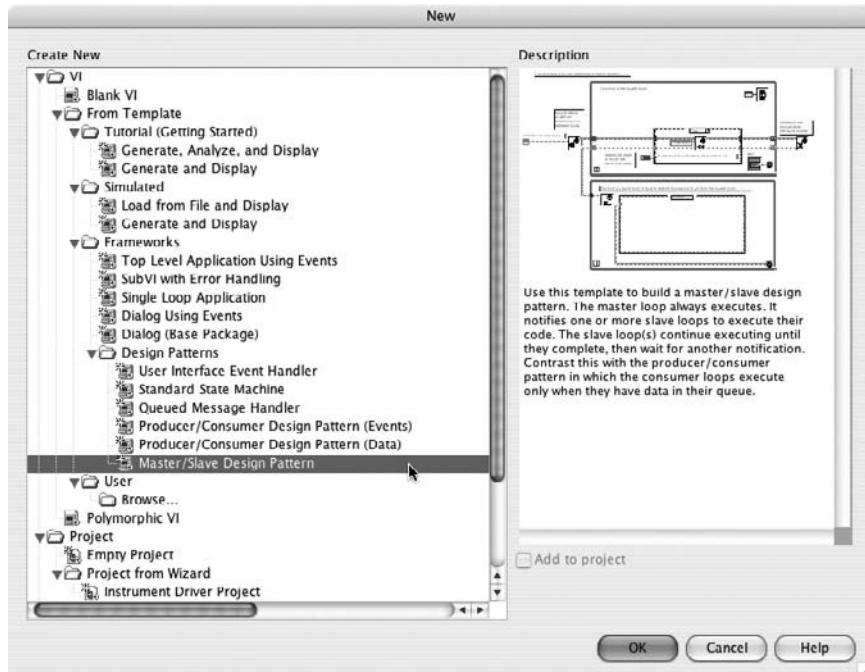
Consider the reusability of the subVIs you create. Can the function be used in several locations in your program? If so, you definitely have a reusable module, saving disk space and memory. If the subVI requirements are *almost* identical in several locations, it’s probably worth writing it in such a way that it becomes a universal solution—perhaps it just needs a mode control. On the other hand, excessive modularity can lead to inefficiency because each subVI adds calling overhead at

execution time (a fraction of a microsecond per VI) and takes up more space on disk.

There are additional advantages to writing a modular program. First, a simple diagram is easier to understand. Other programmers will be able to figure out a diagram that has a few subVIs much more readily than a complex diagram with level upon level of nested loops and Sequence structures. This is like writing a 10,000-line main program in a procedural language. Second, it is generally easier to modify a modular program. For instance, you might want to change from one brand of digital voltmeter to another. By incorporating well-written, modular drivers, you would just substitute brand X for brand Y and be running again in minutes. One goal in LabVIEW programming is to make your diagrams fit on a standard screen (whatever size is most common for computers like yours). This forces you to design and lay out diagrams in a thoughtful manner. Another goal is to limit the diagram memory usage of a subVI to about 500K. Use the **VI Properties** command from the File menu to see how big each component of a VI is. These are not strict rules, however. Main VIs with complex user interfaces, especially those with many Property nodes, invariably take up much screen space and memory.

### Choose an architecture: Design patterns

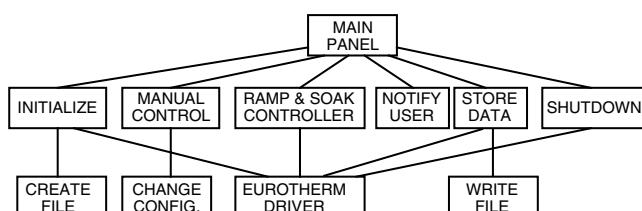
Your initial objective is to decide on an overall **architecture**, or programming strategy, which determines how you want your application to work in the broadest sense. LabVIEW has a few of the most common programming models, known as **design patterns**, built in as templates accessible from the File >> New ... dialog. (Figure 8.7) A design pattern represents the fundamental structure of common LabVIEW applications. One benefit of using a design pattern is that the architecture is already proved to work—there is no need to reinvent the wheel. Plus, when you see a design pattern you've used before in another application, you immediately recognize and understand it. The list of design patterns continues to grow as the years go by. Choosing one is a mix of analysis, intuition, and experience. In Chapter 3, “Controlling Program Flow,” we discussed several design patterns. Study them, and see if your next application doesn't go together a little more smoothly. Although design patterns lay out a common programming structure you can start from, they are not intended to be complete applications. The one aspect missing from almost all the design patterns is proper error handling. They only show you how to use one specific error to terminate parallel loops. They don't even include an error handler like Simple Error Handler.VI on the diagram! See the “Handling Errors” section later in this chapter for techniques you can use to handle errors.



**Figure 8.7** LabVIEW design patterns provide a familiar starting point for many applications. You can access design patterns from the File >> New ... dialog.

### The VI hierarchy as a design tool

A good way to start mapping out your application is to sketch a preliminary **VI hierarchy**. To see what this might look like, open a LabVIEW example VI and select **Show VI Hierarchy** from the Browse menu. You don't need to start off with lots of detail. Just scribble out a map showing the way you would like things to flow and the likely interactions between various subVIs, as in Figure 8.8. Modular decomposition in many languages looks like this figure. There's a main program at the top, several major tasks in the middle level, and system support or



**Figure 8.8** A preliminary VI hierarchy for VBL. At this point, I really didn't know what was going to happen inside each subVI, but at least I had an idea about the division of labor.

I/O drivers at the lowest level. You could also draw this as a nested list, much as file systems are often diagrammed.

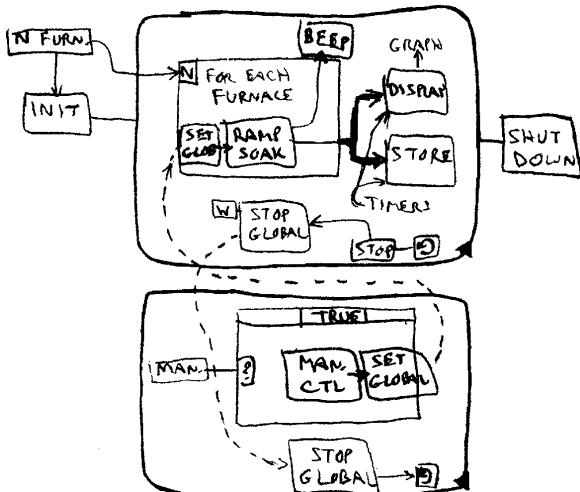
Each **node** or item in the hierarchy represents a task, which then becomes a subVI. Use your list of functional requirements as a checklist to see that each feature has a home somewhere in the hierarchy. As you add features, you can keep adding nodes to this main sketch, or make separate sketches for each major task. Modularity again! There is no need to worry about the programming details inside a low-level task; just make sure you know what it needs to do in a general way. Remember that the hierarchy is a design tool that should be referred to and updated continuously as you write your programs.

The ramp-and-soak controller (one of the major subVIs in VBL) looked simple enough at first, but turned out to be one of the trickiest routines I've ever written. Without a healthy respect for modularity, I would have had a much more difficult time. The problems stemmed from interactions between various modes of operation, such as manual control, ramping, and soaking. My hierarchy sketch was a real mess by the time I was through, but it was the only easy way to track those modes and interactions.

### Sketching program structure

There is a really powerful, visual tool to design LabVIEW programs. It's called **LabVIEW sketching**, and it requires sophisticated hardware: *a pencil and paper*. The idea is to "think in LabVIEW" and put your conceptual program design on paper. You can work at any level of detail. At first, you'll be attacking the main VI, trying to make the major steps happen in the right order. Later on, you will need to figure out exactly where the data comes from, how it needs to be processed, and where it needs to go. Representing this information graphically is a powerful notion. What's even better is that you can implement these sketches directly as LabVIEW diagrams without an intermediate translation step. Figure 8.9 shows Gary's first sketch of the VBL application.

You can also do sketching on your computer with LabVIEW. This has the advantage of permitting you to try out programming techniques that might simplify the problem. Then your test diagram can be pasted right into the real VI. I do this a lot when I'm not sure how a built-in function works. I hook it into a simple test VI and fiddle with it until I really understand. It's a much better approach than to bury a mysterious function deep inside an important VI, then to figure out why things aren't working later on. This scheme works great with top-down design where you have *stub* VIs without diagrams. You simply paste the final results of your experimental development into the diagram of the stub VI.



**Figure 8.9** Sketch for the main VI used in VBL. Not very detailed at this point, but it sure gets the point across.

Show your sketches to other LabVIEW users. Exchange ideas as you do the design. It's a vastly underused resource, this synergy thing. Everyone has something to contribute—a new way of looking at the problem, a trick she or he heard about, or sometimes a ready-to-use VI that fills the bill. To quote from Brad Hedstrom of PE Biosystems, “If you can’t draw it with a pencil and paper, how can you draw it in G?”

### Pseudocoding

If you are an experienced programmer, you may find that it’s easier to write some parts of your program in a procedural language or **pseudocode**. Of course, you should do this only if it comes naturally. Translating Pascalese to LabVIEW may or may not be obvious or efficient, but Gary sometimes uses this technique instead of LabVIEW sketching when he has a really tough numerical algorithm to hammer out. Figure 8.10 shows a likely mapping between some example pseudocode and a LabVIEW data structure, and Figure 8.11 shows the associated LabVIEW diagram that is a translation of the same code.

There’s something bothersome about this translation process. The folks who thought up LabVIEW in the first place were trying to free us from the necessity of writing procedural code with all its unforgiving syntactical rules. Normally, LabVIEW makes complicated things simple, but sometimes it also makes simple things complicated. Making an oscilloscope into a spectrum analyzer using virtual instruments is really easy, but making a character-by-character string parser is a

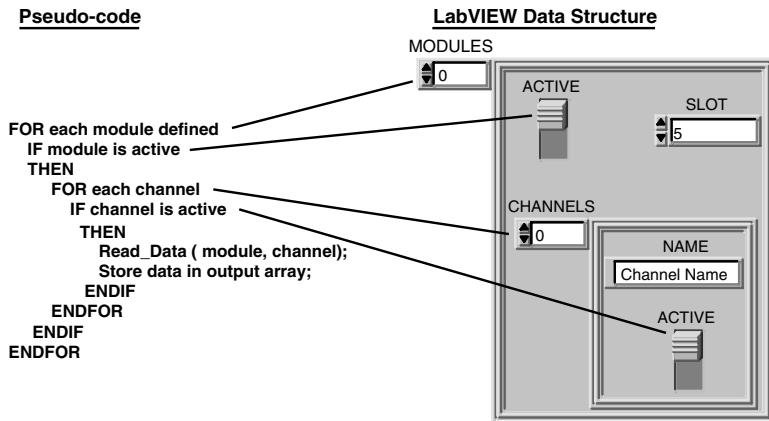


Figure 8.10 A mapping between a pseudocode segment and a LabVIEW data structure.

real mess. In such cases, we choose the best tool for the job at hand and be happy.

To design the difficult ramp-and-soak control logic in VBL, I found that a pseudocode approach was easier. I didn't have to look far to recover the example that appears in Figure 8.12; it was pasted into a diagram string constant in the subVI called *Recipe Control Logic*. I won't bother to reproduce the entire VI diagram here because it has too many nested Case structures to print in a reasonable space.

### Ranges, coercion, and default values

#### CLAD

Protect yourself—expect users to supply ridiculous inputs at every possible opportunity. If your program does not self-destruct under such abuse, then it is said to be **robust**. Unfortunately, quick-and-dirty test

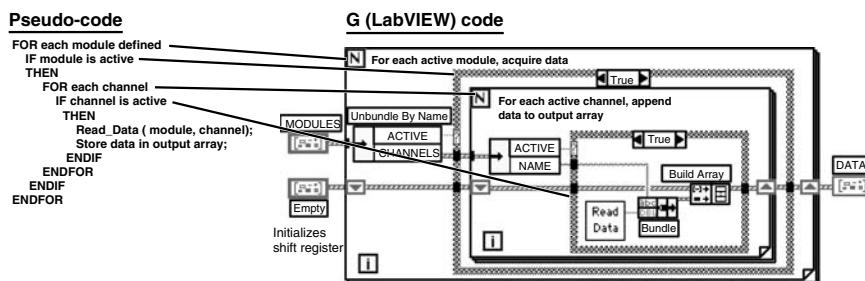
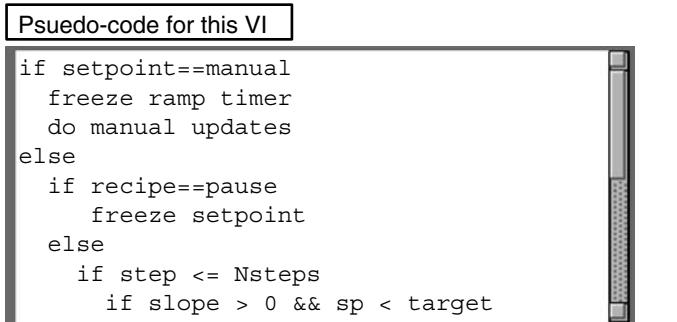


Figure 8.11 The same pseudocode translated into a LabVIEW diagram, which acts on the data structure from the previous figure. If this is your cup of tea, by all means use this approach.

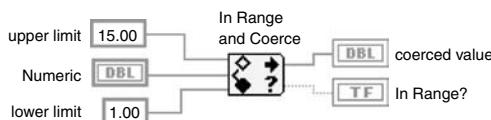


**Figure 8.12** If you do decide to use pseudocode to design your program, include it right on the diagram by pasting it into a String constant like this.

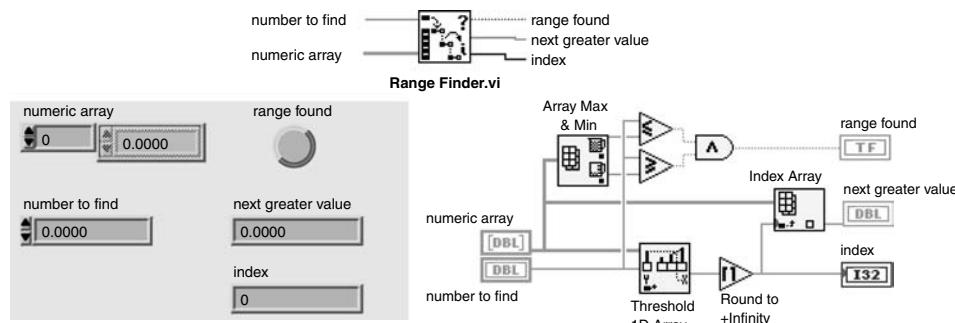
programs are rarely robust. Commercial applications such as word processors are highly robust, lest the publisher go out of business.

You may need to put appreciable effort into bullet-proofing your LabVIEW application against wayward users. On older versions of LabVIEW you could use the **Data Range ...** pop-up dialog to *coerce* values into the desired range (minimum, maximum, and increment). This was a great way to limit inputs to reasonable values, but there was no way to tell if a control had Data Range limits set just by looking at the control. You could get some really strange behavior and spend a lot of time tracking it down to an invisible range coercion on a control or a subVI. As a result, setting the Data Range is no longer allowed for subVIs. Now you will need to use the **In Range and Coerce Function** (Figure 8.13) that applies limits programmatically. If the values are not evenly spaced (such as a 1-2-5 sequence), use a function similar to the **Range Finder VI** shown in Figure 8.14.

Many GPIB instruments limit the permissible settings of one control based upon the settings of another: A voltmeter might permit a range setting of 2000 V for dc, but only 1000 V for ac. If the affected controls (e.g., Range and Mode) reside in the same VI, put the interlock logic there. If one or more of the controls are not readily available, you can request the present settings from the instrument to make sure that you don't ask for an invalid combination.



**Figure 8.13** The In Range and Coerce Function limits the maximum and minimum values of a scalar.



**Figure 8.14** A Range Finder utility VI. This one looks through an array of permissible values and finds the one that is greater than or equal to the value you wish to find. It returns that value and the index into the number to find the array.

String controls and tables are particularly obnoxious because the user can (and will) type just about anything into one. Think about the consequences of this for every string in your program. You may need to apply a filter of some type to remove or replace unacceptable characters. For instance, other applications may not like a data file in which the name of a signal has embedded blanks or nonalphanumeric characters.

One nefarious character is the carriage return or **end-of-line (EOL)** character for your platform. Users sometimes hit the Return key rather than the Enter key to complete the entry of text into a LabVIEW string control, thus appending EOL to the string. Note that training users will not solve the problem! You have two options. In the LabVIEW Preferences, under Front Panel, there is an option to treat Return the same as Enter in string controls. If selected, the two keys act the same and no EOL character is stored. The other alternative is to write a subVI to search for EOL and kill it. The Preferences method is nice, but what happens when you install your VI on another machine? Will that preference be properly set? And what about all the other preferences that you so carefully tuned for your application? If a setting is very important, you must explicitly update the preferences when installing your VI.

Controls should have reasonable default values. It's nice to have a VI that does not fail when run *as-opened* with default values. After entering the values you desire, you select **Make Current Value Default** from the control's pop-up or from the Edit menu. Remember to show the default value and units in parentheses in the control's label so that it appears in the Help window. It's nice to know that the default value is 1000, and whether it's milliseconds or seconds. But don't make the data in graphs and arrays into default values (unless it's really required); that just wastes disk space when the VI is saved.

Make intelligent use of default values. In the case of a utility VI Write Characters to File, the default path is empty, which forces a file dialog. This can save the use of a boolean switch in many cases. With numeric controls, you can assign a default value that is unlikely to be supplied during normal use of the VI and then test for that value. A large negative integer may make sense; or if your input is a floating-point type, enter **NaN** (not a number), and make that the default. In the Comparison functions, you can use **Not A Number/Path/Refnum** to check for NaN. Similarly, empty strings can be tested with the **Empty String/Path** function.

## Handling errors

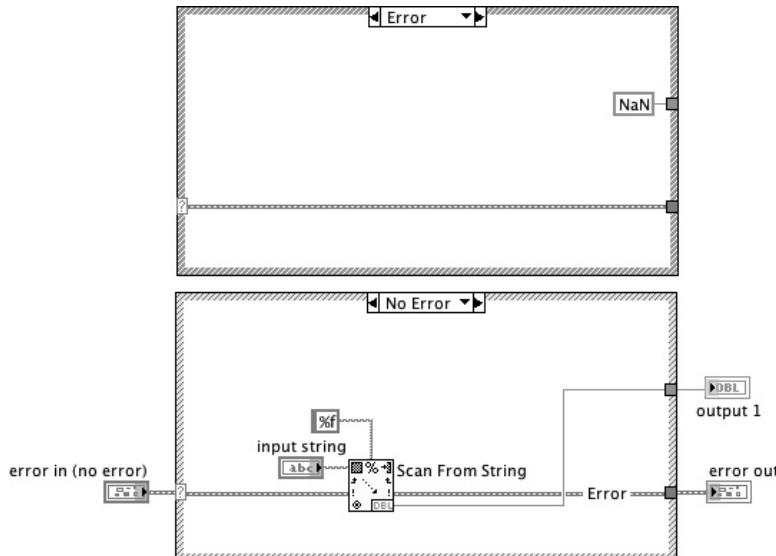
**CLAD**

**CLD**

Another aspect of robust programs is the way they report run-time errors. Commercial software contains an incredible amount of code devoted to error handling, so don't feel silly when one-half of your diagram is devoted to testing values and generating error clusters or dialogs. Most problems occur during the initial setup of a system, but may rarely occur thereafter. It may seem wasteful to have so much code lying around that is used only 0.01 percent of the time. But it's in the start-up phase that users need the most information about what is going wrong. Put in the error routines and *leave them there*. If you want, add a boolean control to turn off dialog generation. Realize that dialog boxes tie up the VI from which they are called until the user responds. This is not a good mode of operation for an unattended system, but you may still want the option of viewing dialogs during a troubleshooting session.

There is a distinction between error handling and error recovery. Error handling provides a graceful method to report an error condition, and to allow the operator to intervene, without causing more errors. A simple message "No scope at GPIB address 5" will go a long way toward getting the problem solved. Error recovery involves programmatically making decisions about the error and trying to correct it. Serious problems can occur when error recovery goes awry, so think carefully before making your application so smart it "fixes" itself. The best applications will report any and all error messages to guarantee system reliability and data quality. Your job during the development phase is not to ignore errors, but to eliminate the conditions that caused an error when it crops up.

The simplest and recommended method to implement error handling in your application is to use an error I/O cluster to connect one VI to the next. The error I/O cluster works as a common dataflow thread, as you have already seen in many examples in this book. The error cluster enforces dataflow and allows VIs downstream to take appropriate action.



**Figure 8.15** Test the error cluster inside each subVI. If there is an input error, no work is performed by the subVI and the error is passed through without modification. The Scan From String function does this internally, as do almost all LabVIEW functions.

It doesn't make sense to work on bad data, or send commands to an unresponsive instrument. Figure 8.15 shows how to implement error handling in a subVI. If there is an input error, the error cluster is passed unmodified to the next VI in the chain. No work is performed on any input error. Internally, this is the default behavior of almost all LabVIEW functions and subVIs. Notable exceptions are functions that close file I/O or instrument sessions. They always attempt to close the I/O before passing along the error message.

At the end of every chain of I/O operations you should place an **error handler** to alert users of any errors. Some common reasons for failure in I/O operations are that

- A system configuration error has been made. For instance, some equipment has not been properly installed or connected. Also, with plug-in boards, you must install the correct driver in your operating system.
- Improper parameters have been supplied, such as channel numbers or file path names.
- A programming error has occurred on the diagram. Typically, you forgot to wire an important parameter.
- LabVIEW bugs crop up (incredible, but true).

A good error handler tells the user the following:

- Where the error came from
- What might have caused the error
- What actions he or she might take to fix and/or avoid recurrence
- One or more appropriate options for escape (continue, stop, abort, etc.)

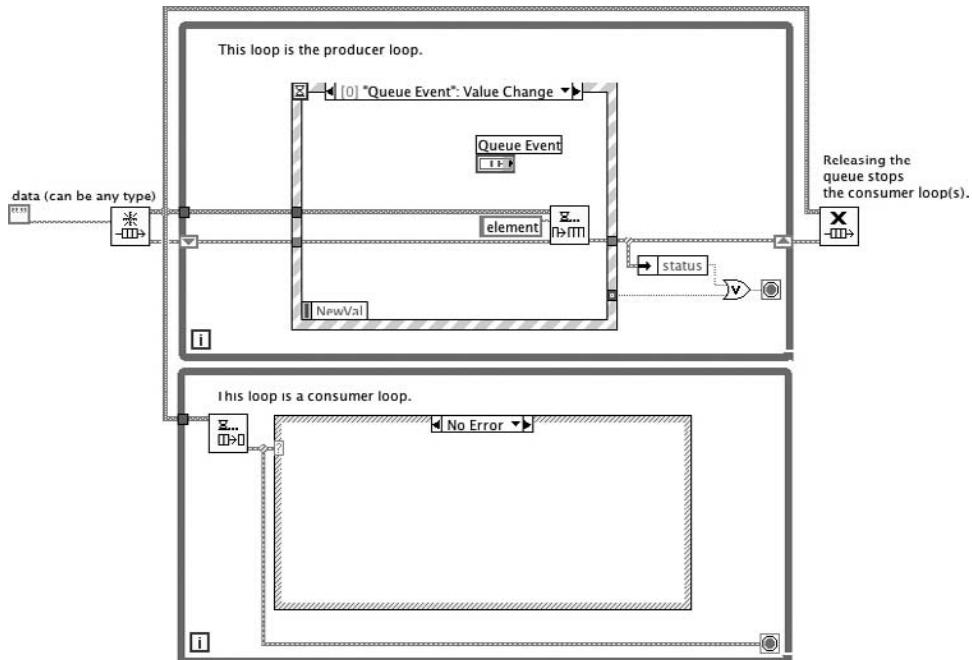
Errors should always be reported in plain language, although an error code number is usually displayed in addition, in case a system guru wants to dig deeper. Dialog boxes are a logical choice when the error is severe enough that user intervention is required. However, avoid burying dialog boxes deep within the VI hierarchy. Dialogs are best placed at the end of an error chain higher in the application layer.

Error handler VIs are included with LabVIEW in the Dialog and User Interface functions. A favorite is called **Simple Error Handler**, and it works for just about any purpose. You can feed it an error code number or a standard error I/O cluster, and it will generate a formatted message and an optional dialog.

Error I/O is essential in LabVIEW programming because it solves one of those universal needs, and as such, the file I/O functions, data acquisition library, and all the newer driver VIs use it. If the error handlers supplied don't suit your needs, use them as a foundation for your own customized version. The **General Error Handler VI** permits you to enter custom error codes and messages. For complex applications you can add custom error codes between the range of 5000 and 9999 into LabVIEW with the **Error Code File Editor** (Tools >> Advanced >> Edit Error Codes...). Custom error codes are saved in the labview\user.lib\errors directory. Throughout the application hierarchy, you can then generate your own error codes, and LabVIEW's error handlers will display your plain language message.

If an error condition is severe enough that execution of the VI must not continue, then shut down the application gracefully. Aborting execution can be risky if any I/O activity is pending. Some kinds of buffered data acquisition operations keep going, and analog and digital outputs retain their last values. It may be difficult to clear up these conditions. Expect the worst and provide a way to programmatically exit on an error.

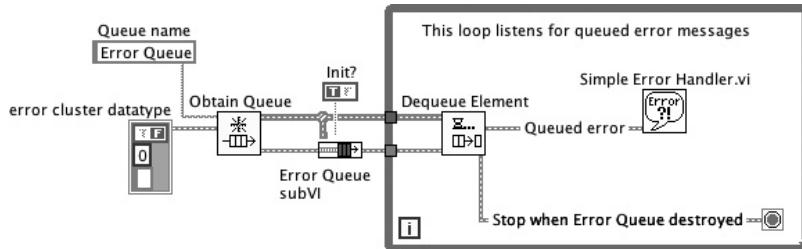
There is a fundamental flaw in the LabVIEW design patterns—the error handling is incomplete. That is so because the design patterns are illustrations to show how to pass messages and programmatically stop one loop from another without using globals. Figure 8.16 shows the stock **producer/consumer (events)** design pattern shipped with LabVIEW 8.0. Sometimes this example is called a **queued message handler**. The top loop produces messages each time the event structure fires.



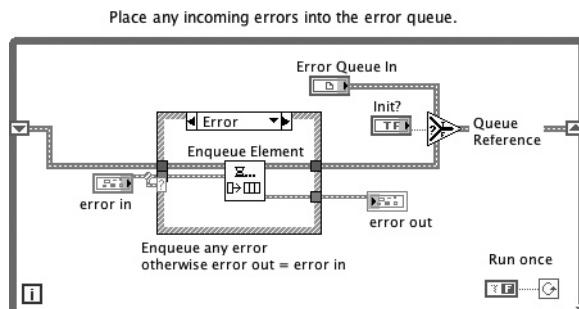
**Figure 8.16** Error handling in producer/consumer design pattern. Any error terminates a loop. Is that what you want in a robust application?

The messages are put into a queue (**enqueued**) by the **producer** and removed from the queue (**dequeued**) by the bottom **consumer** loop. Stopping the producer loop releases the queue and stops the consumer loop when the dequeue function generates an error. If any subVIs in the consumer loop were included in the error chain between the dequeue function and the stop condition, it would programmatically alter the behavior by forcing the loop to terminate on any error. The queued message handler is a common design pattern for programming user-interface applications; however, because there is only one-way communication between the loops, the producer loop will never know if the consumer loop terminated—it will just go on putting messages into the queue. Clearly this is not a robust solution!

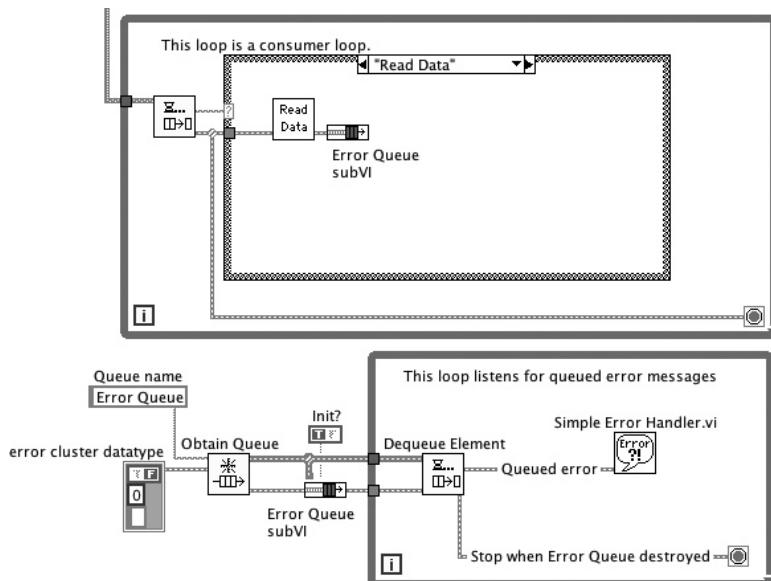
One flexible error-handling scheme we've used, and seen used by others, is based on the addition of an extra top-level error-handling loop to which all error reports are passed via a **global queue**. Any VI is permitted to deposit an error in the queue at any time, which causes the error handler to wake up and evaluate that error. Figure 8.17 shows the top-level error-handling loop. The reference to **Error Queue** is stored in **ErrorQueue.vi** (Figure 8.18). Any incoming error placed into the queue is handled by the **Simple Error Handler.vi**. Figure 8.19 shows one way to put the code into action.



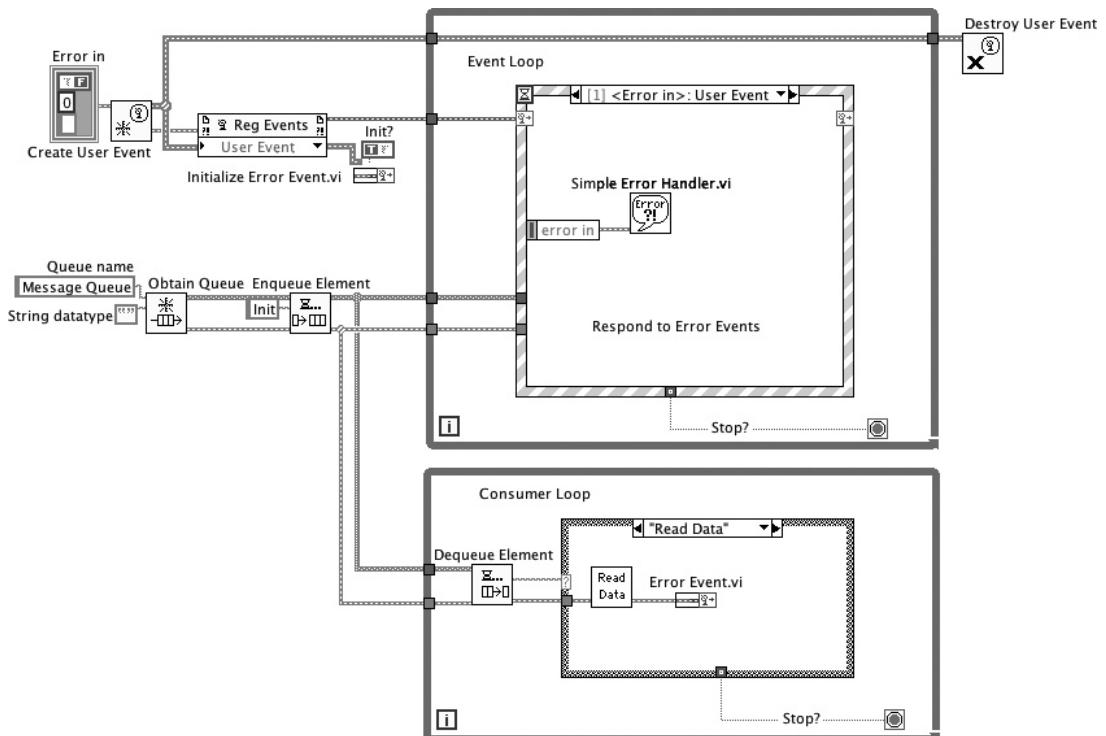
**Figure 8.17** Error loop handles all errors. The Error Queue reference is stored in ErrorQueue.vi (Figure 8.18).



**Figure 8.18** ErrorQueue.vi enqueues incoming errors into the Error Queue. The queue reference is stored in a shift register and remains valid until the queue is destroyed.



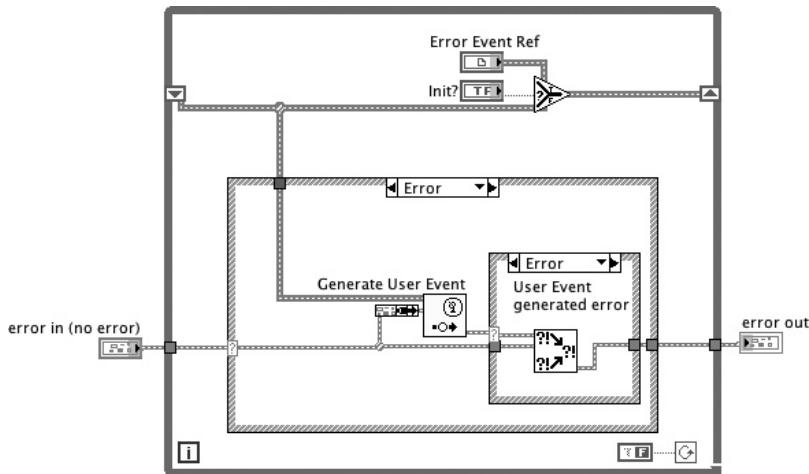
**Figure 8.19** ErrorQueue.vi and the error-handling loop in action. All errors are reported via the Error Queue.



**Figure 8.20** The event structure responds to front panel events and programmatic events. Dynamic, user-defined events pass errors from the consumer loop back to the event loop.

Reporting all the errors via a global queue has a lot of advantages, but we have to add an extra dequeue loop and don't have a way to trigger the event loop. Figure 8.20 shows an advanced technique using **dynamic events** to pass a **user-defined event** to the event loop. Using a single event loop to listen to front panel user events and programmatic user-defined events enables two-way communications between the loops. Figure 8.21 shows the **Error Event.VI**. It is a **functional global** that retains the reference to the dynamic event in a shift register. Any incoming errors fire the user event and pass the error cluster to the **Simple Error Handler.vi** in the Event Loop.

Whichever error-handling method you put into place needs to be robust and flexible enough to grow with your application. Above all, the requirements for your application should dictate what level of error handling or error recovery you need to put in place. The methods we've shown you are by no means the only way to do things. If they don't work for you, get creative and invent your own.



**Figure 8.21** Incoming errors fire the user-defined event, and the error cluster is passed to the waiting Event structure.

VBL had to run reliably when unattended. Therefore, I had to avoid any error dialogs that would lock up the system. When you call one of the dialog subVIs, the calling VI must wait for the user to dismiss the dialog before it can continue execution. That's the reason you should never generate an error dialog from within a driver VI; do it at the top level only. My solution was to add a switch to enable dialog generation only for test purposes. If a communications error occurred during normal operation, the driver would retry the I/O operation several times. If that failed, the main program would simply go on as if nothing had happened. This was a judgment call based on the criticality of the operation. I decided that no harm would come to any of the furnaces because even the worst-case error (total communications failure) would only result in a *really long* constant temperature soak. This has proven to be the correct choice after several years of continuous operation.

### Putting it all together

Keep hammering away at your application, one VI at a time. Use all the tools we have discussed for each step of the process. Remember that each VI can and should be treated as a stand-alone program. Break it down into pieces you understand (more subVIs) and tackle those one at a time. Then build up the hierarchy, testing as you go. It's a reliable path to success. Never write your program as one huge VI—the LabVIEW equivalent of a 40,000-line main program—because it will be very difficult to troubleshoot, let alone understand. Documenting as you go can also save time in the long run because it's easier to write about what you've done while you're doing it. The same is true for quality

assurance documents such as test procedures. If formal testing is required, do a dry run of the formal testing as early as possible in the development process. That means the procedure is complete and any required test software is written and validated.

One of the LabVIEW inventors, Jack MacCrisken, after watching many users go through the development process over the years, is a firm believer in iterative design. His rule of thumb is that every application should be written twice. The first time, it's ugly, but it gets the job done and establishes a viable hierarchical approach. The second time, you sweep through and optimize, clean up, and document each VI, as well as redesign any parts of your program that no longer suit the overall requirements.

Check your user requirements document from time to time to make sure that everything has been addressed. Keep the users in the loop by demonstrating parts of your application as they are completed. It's amazing what an operator will say when shown a LabVIEW panel that mimics real laboratory equipment. You need this feedback at all stages of development as a form of reality check. In the real world of science, requirements change. If the experiment succeeds, the requirements grow. If it produces unexpected results, requirements change. And users change, too! You and your programs need to be flexible and ready to accept change. An excellent article on this subject, written by Ted Brunzie of JPL, appears in *LabVIEW Technical Resource* (1995).

## Testing and Debugging Your Program

If you have designed your program in a modular fashion, testing will *almost* take care of itself: You are able to test each subVI independently, greatly increasing your chances of success. It's comforting to know that each module has been thoroughly thrashed and shows no signs of aberrant behavior. Nonmodular programming will come back to haunt you because a big, complicated diagram is much harder to debug than a collection of smaller, simpler ones.

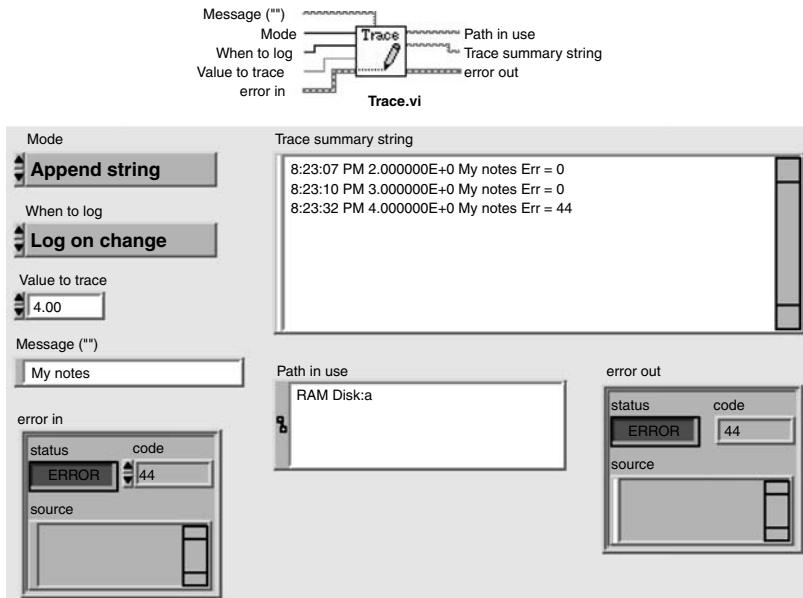
### Tracing execution

Many procedural languages contain a **trace** command or mode of operation, in which you can record a history of values for a given variable. A string indicator on the panel of a VI will suffice when the message is more of an informative nature. A handy technique is to use a functional global with a reference to a queue to pass informative messages. This is very useful when you're not sure about the order in which events occurred or values were computed. An idea Gary got from Dan Snider, who wrote the Motion Toolbox for Parker Compumotor, is a **trace VI**

that you can insert in your program to record important events. Values can be recorded in two ways: to disk or by appending to an array. The VI can record values each time it's called, as a simple datalogger does, or it can watch for changes in a particular value and record only the changes. This implementation is not perfect, however. If you have a race condition on a diagram in which you are writing to a global variable at two locations almost simultaneously, the trace VI may not be quick enough to separate the two updates. This is where a built-in LabVIEW trace command would save the day. Perhaps a future version of LabVIEW will have this feature.

Figure 8.22 shows the connector pane and front panel of the Trace VI. In this example, trace information may be appended to a file or to a string displayed on the panel of the VI. Each trace message contains a timestamp, the value being traced, an optional message string, and the error code. A single numeric value is traced, but you can replace it with whatever data you may have. Edit the diagram to change the formatting of the string as desired.

The **When to log** control determines when a trace record will be logged to the file or string. *Log always* means a record will be saved every time the VI is called. *Log on change* means only record data if



**Figure 8.22** The Trace VI can help debug some applications by recording changes in important variables. You can record to the string indicator on the panel or to a file.

**Value** has changed since the last time the VI was called. The **Mode** control determines what operation to perform. You can clear the string, choose a new file, log to the string, or log to the file. If you are routing trace information to the string, you may open the VI to display the trace summary. We also include a mode called *nothing*, which makes the VI do exactly that. Why include such a mode? So that you can leave the Trace VI in place on a diagram but in a disabled condition.

This VI is usually wired into a diagram to monitor a specific variable. You can also place it in an independent While Loop, perhaps in its own top-level VI, and have it monitor a global variable for any changes. It's important to remember that the Trace VI takes some time to do its job. Appending to the string starts out by taking a few milliseconds, but it gradually slows down as the string grows. Logging to a file will also take some milliseconds, but it's a constant delay. Displaying the panel while logging to the string can seriously degrade performance. You may want to modify the VI for higher performance by stripping out all the extra features you don't need. In particular, to trace a single numeric value, all you have to do is to see if its value has changed, then append it to an array. That's much faster than all the string handling, but it's not quite as informative.

### Checking performance

During development, you should occasionally check the performance of your application to make sure that it meets any specifications for speed or response time. You should also check memory usage and make sure that your customers will have sufficient memory and CPU speed for the application. LabVIEW has a VI performance profiler that makes it much easier to verify speed and memory usage for a hierarchy.

Select **Profile VIs** from the Tools >> Profile >> Performance and Memory... menu to open the Profile Window and start a profiling session. There are check boxes for selecting timing statistics, timing details, and memory usage. It's OK to turn everything on. Click the **Start** button to enable the collection of performance data, and then run your VI and use it in a normal manner. The Profile Window must stay open in the background to collect data. At any time, you can click the **Snapshot** button to view the current statistics. Statistics are presented in a large table, with one row for each VI. You can save this data in a tab-delimited text file by clicking the **Save** button.

Scroll through the table and note the almost overwhelming bounty of information. (See the LabVIEW user's manual for details on the meaning of each value.) Click on any column header to sort the information by that statistic. For instance, you will probably want to know which VIs use the most memory. If you see something surprising in the list,

such as a subVI using a large amount of memory or taking up too much time, it's time to investigate. Note that the memory and timing values are relative—the profiler itself uses some memory and CPU time—but the comparisons are accurate.

## Final Touches

If you follow the directions discussed here, you will end up with a well-designed LabVIEW application that meets the specifications determined early in the process. A few final chores may remain.

**Test and verify** your program. First, compare your application with the original specifications one more time. Make sure that all the needs are fulfilled—controls and indicators are all in place, data files contain the correct formats, throughput is acceptable, and so forth. Second, you should abuse your program in a big way. Try pressing buttons in the wrong order and entering ridiculous values in all controls (your users will!). When data files are involved, go out and modify, delete, or otherwise mangle the files without LabVIEW's knowledge, just to see what happens. Remember that error handling is the name of the game in robust programming. If you are operating under the auspices of formal software quality assurance, you will have to write a detailed validation and verification plan, followed by a report of the test results. A phenomenal amount of paperwork will be generated. Formal software quality assurance methods and software engineering are discussed in detail in *LabVIEW Power Programming* (Johnson 1998).

**Train** your users. Depending on the complexity of the application, this may require anything from a few minutes to several hours. Gary once developed a big general-purpose data acquisition package and had about 15 people to train to use it. The package had lots of features, and most of the users were unfamiliar with LabVIEW. His strategy was to personally train two people at a time in front of a live workstation. The user's manual he wrote was the only other teaching aid. After going through the basic operations section of the manual in order during the training session, the students could see that the manual was a good reference when they had questions later. It took about 2 h for each training session, much of which consisted of demonstration, with hands-on practice by the users at each major step. Simple reinforcement exercises really help students to retain what you have taught them. Everyone really appreciated these training sessions because they helped them get a jump start in the use of the new system.

Last, but not least, make several **backup** copies of all your VIs and associated documents. Hopefully, you have been making some kind of backups for safety's sake all along. It's an important habit to get into because system failures and foul-ups on your part should never become

major setbacks. A good practice is to keep the working copy on your hard disk, a daily backup on another (removable) storage device, and a safety copy on floppies or some other medium, perhaps a file server. When you make up a deliverable package, you can put a set of master floppies or CD-R in one of those clear plastic diskette holders in the back of the user's manual. It looks really "pro."

### VBL epilogue

It turns out that it took longer to get the equipment fabricated and installed in Larry's lab than it did for me to write the application. I did all the testing in my office with a Eurotherm controller next to my Mac, including all the demonstrations. Since there was plenty of time, I finished off the documentation and delivered a copy in advance—Larry was pleasantly surprised. Training was spread out over several sessions, giving him plenty of time to try out all the features. We tested the package thoroughly without power applied to the furnaces for safety's sake. Only a few final modifications and debugging sessions were needed before the package was ready for real brazing runs. It's been running constantly since the end of 1992 with excellent results. Larry is another happy LabVIEW customer.

## Studying for the LabVIEW Certification Exams

LabVIEW Certifications are valuable professional credentials that validate your LabVIEW knowledge and programming skills. Attaining certification ensures your employers and customers that you actually know what you are doing. There are three levels of LabVIEW certification: the basic **Certified LabVIEW Associate Developer (CLAD)**, next is the **Certified LabVIEW Developer (CLD)**, followed by the **Certified LabVIEW Architect (CLA)**. Each level of certification builds on the knowledge required by the previous level. The CLAD and CLD examinations focus on LabVIEW programming skills, while the CLA examination focuses on advanced application development and the skills required to lead a team of LabVIEW developers. Originally the CLAD and CLD were one examination with two parts: a 40-question multiple-choice test for LabVIEW knowledge, followed by a timed application development examination. Today there are two separate examinations, and you have to pass the CLAD before you can register for the CLD.

### CLAD



The CLAD is a 1-h, computer-based multiple-choice examination administered by Pearson Vue. You can register for the CLAD online by following

the links from National Instruments' Web site ([www.ni.com](http://www.ni.com)). Study and apply the material we've highlighted in Chapters 2 through 9, then take the sample examination National Instruments has online, and you should be fine. You can find links to additional study material online ([www.ni.com](http://www.ni.com)). Since the CLAD is a multiple-choice computer-based examination, you get your score immediately; however, you don't get any feedback on what questions you missed. But you shouldn't have any trouble if you understand LabVIEW and how the functions work.

Here's a summary of the CLAD examination topics and some thinking questions to lead you down the path of enlightenment. If you look through Chapters 2 through 9, you'll see that we've covered these in depth.

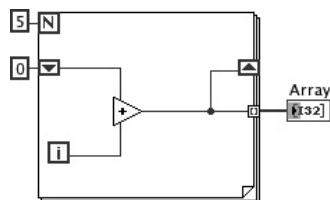
- **LabVIEW programming principles.** Know how to use dataflow and polymorphism in your VIs.
- **LabVIEW environment.** Obviously if you're going to be a LabVIEW programmer, you have to know how to get around and where to find things on menus and palettes. You also want to know the difference between a front panel and a block diagram.
- **Data types.** A LabVIEW programmer needs to know how LabVIEW handles data: the different data types, representations, formats, ranges, scaling, and coercion. What are type definitions and strict type definitions? Know where they are used and how to create them. What is a Data structure?
- **Software constructs.** Know the LabVIEW programming structures and how they differ. What is a shift register and what are some of the ways it can be used? Know which structures index on their boundary and how they differ in function and performance. What are tunnels and sequence locals?
- **Graphical-User Interface (GUI) elements.** Know the different charts and graphs, their display properties, and how to format data for each one. Understand the mechanical actions of booleans. Know when, where, and how to use Property nodes.
- **Variables and functions.** Know the difference between local and global variables, and understand race conditions. Know the LabVIEW timing functions, file I/O functions, and how to work with binary and text files. What is a cluster and how can you bundle or unbundle a single specific element? Understand how LabVIEW works with arrays on loop boundaries and how the array functions operate. How do you pull a single element out of an array? What if it's a 2D array? How do you slice out a single row or column? Know the parts of LabVIEW's waveform data type and how to access them.

- **Simple design patterns.** Understand LabVIEW's multitasking environment and basic design patterns. How does a Case structure work? What is a state machine?
- **SubVI design.** Know about subVI connector panes, icons, and VI configuration options. How do you make connections required or optional, and how can you distinguish between them in the Context Help window?
- **VI design and documentation.** Know how to build VIs and applications according to the *LabVIEW Style Guide* and the *LabVIEW Development Guidelines*. Know how to create VI and control descriptions that show up in the Context Help window. How do you create tip strips? What are the best practices for documenting VIs?
- **Error handling.** Know how error I/O passes error information between VIs, how a subVI should respond to an input error, and the proper terminals for error I/O.
- **Debugging tools and techniques.** How do LabVIEW's debugging tools work and what are some best practices?

That's a lot of stuff to know, but it's basic LabVIEW knowledge you should have if you are going to call yourself a LabVIEW programmer. The test itself is not too hard. The questions are designed to test your knowledge, not to confuse or trick you. Here is our own example of one type of programming problem and question found on the CLAD.

What are the values in Array after the For Loop in Figure 8.23 has finished?

- A) 1, 2, 3, 4, 5
- B) 0, 1, 2, 3, 4
- C) 1, 3, 6, 10, 15
- D) 0, 1, 3, 6, 10



**Figure 8.23** This VI represents one type of program analysis problem you may encounter on the CLAD. What are the values in Array after the For Loop has finished?

In our example you have to do a little addition to get the answer; but if you understood For Loops, shift registers, and arrays, it was easy. The rest of the examination is practical questions on the subjects we covered in this book.

**CLD****CLD**

The Certified LabVIEW Developer examination is a 4-h test of your LabVIEW application development skills. You take the examination at a National Instruments-approved location on a PC with a current version of LabVIEW installed. You are not allowed to use any toolkits or outside LabVIEW code other than the templates and examples that normally come with LabVIEW. All that you are given is a sealed envelope with two things: a blank floppy for your completed application, and a multipage document with application requirements and functional description(s). And 240 min later you should have a completed application that

- Functions as specified. It is always important to have an application do what the customer wants.
- Conforms to LabVIEW coding style and documentation standards. The *LabVIEW Development Guidelines* is available in the online help if you need to refer to it.
- Is created expressly for the examination using VIs and functions available in LabVIEW.
- Is hierarchical. All major functions should be performed in subVIs.
- Uses a state machine. Your state machine should use a type-defined enumerated control, a queue, or an Event structure for state management.
- Is easily scalable to more states and/or features without having to manually update the hierarchy. Think typedef enum here.
- Minimizes the use of excessive structures, variables (locals/globals), and Property nodes.
- Responds to front panel controls (within 100 ms) and does not use 100 percent of the CPU cycles.
- Closes all opened references and handles.
- Is well documented and includes
  - Labels on appropriate wires within the main VI and subVIs
  - Descriptions for each algorithm
  - Documentation in VI Properties >> Documentation for the main VI and subVIs

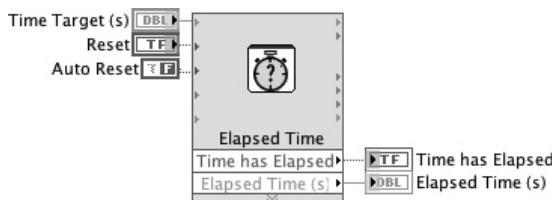
- Tip strip and descriptions for front panel controls and indicators
- Labels on all constants

The CLD examination is graded on three criteria:

- **Functionality.** Does the application do what was asked? Is the logic correct? Does the VI respond to front panel events within the time limit? Does the application produce errors? Is file I/O implemented correctly?
- **Style.** Is the application within professional LabVIEW programming guidelines? Is it neat? Is it scalable and maintainable? Does it make excessive use of locals, globals, and Property nodes? Does it use all the CPU time?
- **Documentation.** Place labels on long wires. Algorithms should be documented with a text label on the block diagram. VIs should have descriptions. Front panel controls and indicators should have tip strips and descriptions. VIs should have descriptive icons. All controls, indicators, and constants should have descriptive names.

There is a total of 40 points. Functionality counts 15 points, style counts 15 points, and documentation counts 10 points. You pass if you score 75 percent or more.

You have a lot to do in 4 h; if you're the kind of programmer who likes to go away and think about a problem before coming up with an elegant solution, you may find yourself running out of time. Sometimes simple solutions turn out to be the best. One function that we think you'll find handy in many situations is the **Elapsed Time Express** VI (Figure 8.24) available from the Programming >> Timing palette. The first time you call it, you will need to **Reset** it and give it a **Time Target (s)**. The next iteration it will return **Elapsed Time** in seconds and a boolean indicating whether **Time Has Elapsed**. We recommend spending some time with this VI and getting to know it well. One idiosyncrasy of this function is caused by the internal use of the **First Call?** primitive to reset itself. Attempting to iteratively develop and debug a subVI built



**Figure 8.24** The Elapsed Time Express VI is a handy function in many situations.

around this function can drive you nuts because Elapsed Time will reset itself each time you push the subVI's Run button.

Now that you know what to expect, we're going to give you three sample examinations to work through. These are retired CLD exams, so they are what you can really expect to see when you take the examination. Remember, you only have 4 h from the time you see the examination to deliver a documented, scalable, working application that is data-flow-driven and doesn't use a ton of locals or Property nodes. Are you ready? On your mark, get set, Go!

### Example 1: Traffic light controller

**Task:** Your company has received a contract to produce a traffic light controller for a local city government. The purpose of the traffic light controller is to maintain traffic at a heavily used 4-way intersection.

You have been assigned to create the software for this system using LabVIEW. You will create the software for one intersection that includes north, south, east, and west bound traffic, with each direction having a left-hand turn lane (Figure 8.25).

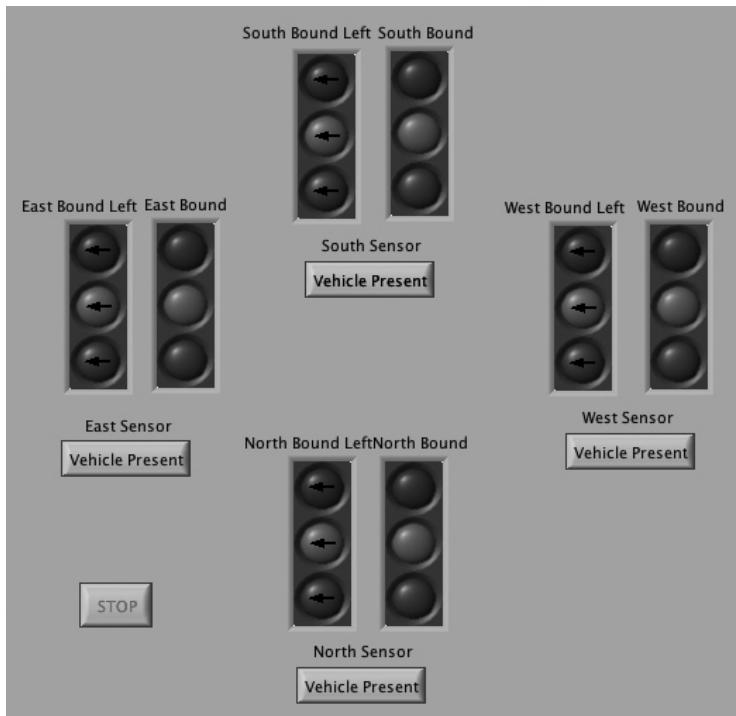


Figure 8.25 Front panel for the traffic light controller.

**TABLE 8.1 Traffic Light Pattern with No Left-Turn Sequence**

Northbound	Southbound	Eastbound	Westbound
Green	Green	Red	Red
Yellow	Yellow	Red	Red
Red	Red	Green	Green
Red	Red	Yellow	Yellow

**General operation.** The traffic light controller provides for green, yellow, and red lights for each direction. In addition, a second green-yellow-red light must be provided for the left-hand turn lane.

The periods of time the lights should be on are as follows:

**Red:** 4 s

**Yellow:** 2 s

**Green:** 4 s

The pattern of the lights with no left-turn sequence should be according to Table 8.1.

The pattern of lights with a left-turn sequence for an eastbound vehicle should be according to Table 8.2.

The left-turn sequence should be inserted only if a vehicle is positioned over the left-turn lane sensor at the time the left-turn sequence should start.

### Example 2: Car wash controller

**Task.** Your company has received a contract to design a controller for a touchless car wash. A touchless car wash is one that relies on water spray only—there are no brushes. The specifications and rules are listed below. Your job is to design a controller using LabVIEW that satisfies all specifications.

**Instructions.** Using a front panel similar to the graphic provided in Figure 8.26, create a LabVIEW application that implements the actions

**TABLE 8.2 Traffic Light Pattern with an Eastbound Turn Sequence**

Northbound	Southbound	Eastbound	Westbound
Green	Green	Red	Red
Yellow	Yellow	Red	Red
Red	Red	Left Green	Left Green
Red	Red	Left Yellow	Left Yellow
Red	Red	Green	Green
Red	Red	Yellow	Yellow



**Figure 8.26** Front panel for the car wash controller.

of the procedure. For this application, the car wash switches are simulated by switches on the front panel. The car wash starts when a purchase switch has been selected. The execution of a cycle is denoted by illuminating the appropriate light-emitting diode (LED). For this application, each cycle's duration should be set to 5 s unless otherwise stated in the Operating Rules below.

#### Car wash specifications

##### Purchase options

- \$5 Deluxe wash
- \$3 Economy wash

##### Cycles and associated timing:

1. Underbody wash, 10 s
2. Soap application, 5 s
3. Main high-pressure wash, 5 s
4. Clear water rinse, 5 s
5. Air dry cycle, 10 s

##### Car wash operating rules

1. Only one purchase selection can be made at a time. The car wash should not accept a second purchase selection while the car wash is in operation.

2. Not all cycles are performed for each wash. The more expensive wash performs more cycles. The cycles performed include
  - Cycles performed for the deluxe wash: 1 - 2 - 3 - 4 - 5
  - Cycles performed for the economy wash: 2 - 3 - 4
3. Each cycle is initiated by a switch. If the vehicle rolls off of the switch, the wash immediately pauses and illuminates an indication to the driver to reposition the vehicle. The amount of time that expires while the car is out of position should not count against the wash time.
4. Underbody wash cycle: The spray heads for this wash are located near the entrance of the car wash. The underbody spray heads are fixed in position, and they require the vehicle to slowly drive over them to wash the underbody of the vehicle. The underbody wash is activated under the following conditions:
  - a. The deluxe wash has been purchased.
  - b. The car wash is in the underbody wash cycle.
  - c. Under Body Wash Position Switch is closed (proximity switch). This cycle of the wash should last for 10 s. Upon completion of this cycle the controller should signal moving to the next cycle by activating the Vehicle Out of Position LED.
5. Main wash cycle: Main Wash Position Switch verifies the vehicle is in the correct location for the wash cycles (cycles 2, 3, and 4) to operate. Each cycle should last for 5 s. If the vehicle rolls off of the Main Wash Position Switch, the wash immediately pauses and illuminates an indication to the driver to reposition the vehicle. The amount of time that expires while the car is out of position should not count against the wash time. The wash resumes after the vehicle is properly positioned. Upon completion of this cycle the controller should signal moving to the next cycle by activating the Vehicle Out of Position LED.
6. Air dry cycle: The air drier is a set of fixed-position blowers located near the exit of the car wash. They require the vehicle to drive slowly out of the car wash through the airstream to dry the vehicle. The air dry cycle activates on the following conditions:
  - a. The deluxe wash has been purchased.
  - b. The car wash has reached the air dry cycle.
  - c. Air Dry Position Switch is closed (proximity switch).

If the vehicle rolls off of the Air Dry Position Switch, the wash immediately pauses and illuminates an indication to the driver to

reposition the vehicle. The amount of time that expires while the car is out of position should not count against the air dry time. The wash resumes after the vehicle is properly positioned. This cycle of the wash should last for 10 s. Upon completion of this cycle the controller should allow the next vehicle in line to select another wash.

7. The car wash must respond to the STOP boolean and Vehicle Position Switches within 100 ms. The STOP boolean aborts the operation of the VI.

### Example 3: Security system

**Task.** Your company has received a contract to produce a security system for a local manufacturer. The purpose of the security system is to monitor the perimeter of the manufacturing complex to prevent unauthorized personnel from entering the grounds. Your job is to design a security system using LabVIEW that satisfies the provided specifications.

**Instructions.** Using a front panel similar to the graphic provided, create a LabVIEW application that implements the actions of the security system (Figure 8.27). You will create the software for six

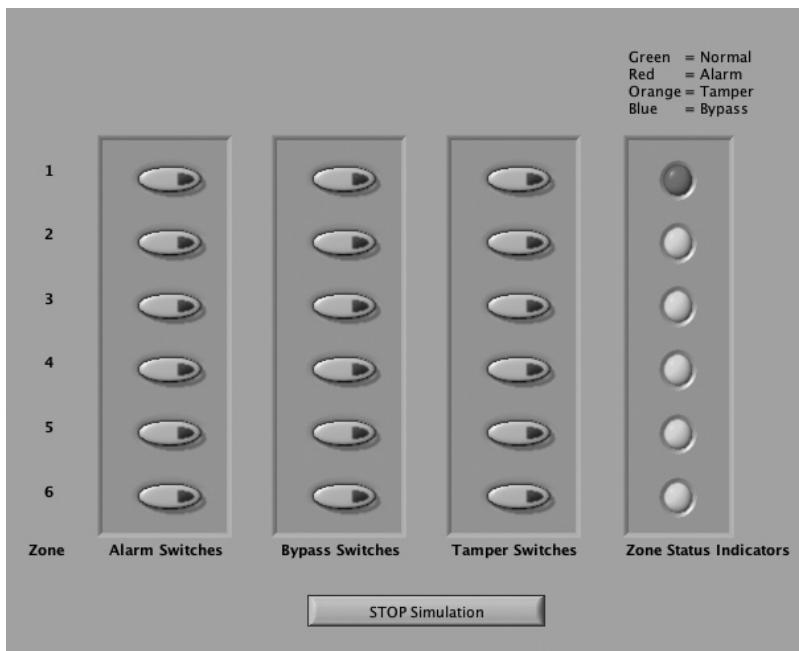


Figure 8.27 Front panel for the security system.

perimeter zones, with the actual hardware being simulated by boolean switches on the front panel. The application should be easily scalable to add more zones without having to manually update the hierarchy.

### Definitions

**Zone:** A perimeter area with its own security status indication.

**Alarm:** A zone condition indicating an intrusion into a zone.

**Bypass:** A state in which a zone will not indicate an alarm condition. Placing a zone in bypass prevents nuisance alarms when maintenance work is performed in the area near the perimeter fence, or when severe weather may cause false alarms.

**Tamper:** A condition where the wiring of a zone has been altered in some way. All zone wiring is supervised. That is, if an individual attempts to circumvent the system by altering the wiring, that zone will indicate a tamper condition.

**Description of controls/indicators.** The security system software accepts boolean inputs, via front panel switches (one set for each zone) for the following functions:

- Alarm input
- Tamper input
- Bypass input

The security system provides for one indicator light for each zone. The color of the light provides the status information for that zone. The colors are

- Green: normal
- Red: alarm
- Blue: bypass
- Orange: tamper

### Operation of the security system

#### System

- A bypass input should always override an alarm input but should not turn off an existing alarm condition. An alarm condition should not be indicated while a zone is in a bypass condition.
- A tamper input should always override both an alarm input and/or a bypass input but should not turn off existing alarm and/or

**TABLE 8.3 Security System Log Format**

Date	Time	Zone	Status
XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX
XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX

bypass conditions. Alarms and bypass conditions should not be indicated while a zone is in a tamper condition.

- An existing condition should be indicated when an overriding input is removed.

**File logging.** The security system should produce an ASCII disk file in a spreadsheet-readable format. The data, when imported into a spreadsheet, should be in the format shown in Table 8.3, where XXXX represents logged data.

The Status field should contain a string indicating the condition of the zone: normal, larm, Bypass, or Tamper.

The log file should be stored in a location relative to the location of the security application and should be updated only when the status of a zone changes and closes after every transaction.

## Bibliography

- Bonal, David: "Mastering the National Instruments Certification Exams," NIWeek 2005, National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2005.
- Brunzie, Ted J.: "Aging Gracefully: Writing Software that Takes Changes in Stride," *LabVIEW Technical Resource*, vol. 3, no. 4, Fall 1995.
- Fowler, Gregg: "Interactive Architectures Revisited," *LabVIEW Technical Resource*, vol. 4, no. 2, Spring 1996.
- Gruggett, Lynda, "Getting Your Priorities Straight," *LabVIEW Technical Resource*, vol. 1, no. 2, Summer 1993. (Back issues are available from LTR Publishing.)
- Johnson, Gary W. (Ed.): *LabVIEW Power Programming*, McGraw-Hill, New York, 1998.
- Ritter, David: *LabVIEW GUI*, McGraw-Hill, New York, 2001.
- Sample Exam, "Certified LabVIEW Developer Examination—Car Wash Controller," National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2004.
- Sample Exam, "Certified LabVIEW Developer Examination—Security System," National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2004.
- Sample Exam, "Certified LabVIEW Developer Examination—Traffic Light Controller," National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2004.

*This page intentionally left blank*

## Documentation

Most programmers hold documentation in the same regard as root canal surgery. Meanwhile, users hold documentation dear to their hearts. The only resolution is for you, the LabVIEW programmer, to make a concerted effort to get that vital documentation on disk and perhaps on paper. This chapter describes some of the documentation tools and formats that have been accepted in commercial LabVIEW software.

The key to good documentation is to generate it as you go. For instance, when you finish constructing a new subVI, fill in the Get Info VI description item, described later. Then when you want to put together a software maintenance document, you can just copy and paste that information into your word processor. It's much easier to explain the function of a VI or control when you have just finished working on it. Come back in a month, and you won't remember any of the details.

If you want to write commercial-grade LabVIEW applications, you will find that documentation is at least as important as having a well-designed program. As a benchmark, budget 25 percent of your time on a given contract for entering information into various parts of VIs and the production of the final document.

### VI Descriptions

**CLAD**

The VI description in the **Documentation** dialog box from the VI Properties is often a user's only source of information about a VI. Think about the way that you find out how someone else's VIs work, or even how the ones in the LabVIEW libraries work. Isn't it nice when there's

online help, so that you don't have to dig out the manual? Important items to include in the description are

- An overview of the VI's function, followed by as many details about the operation of the VI as you can supply
- Instructions for use
- Description of inputs and outputs
- List of global variables and files accessed
- Author's name and date

Information can be cut and pasted into and out of this window. If you need to create a formal document, you can copy the contents of this description into your document. This is how you manage to deliver a really nice-looking document with each driver that you write without doing lots of extra work. You just write the description as if you were writing that formal document. You can also use the techniques described in the following section on formal documents to extract these comments automatically.

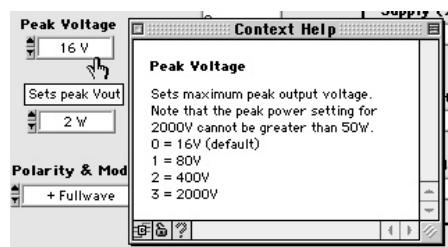
You can also display the VI description by

- Selecting the Connector Pane and Description option when using the Print command from the File menu
- Showing the Context Help window and placing the wiring tool on a subVI's icon on the block diagram

## Control Descriptions

### CLAD

Every control and indicator should have a description entered through the **Description and Tip** pop-up menu. You can display this information by showing the Context Help window and placing the cursor on the control (Figure 9.1). You can also display it by placing the wiring



**Figure 9.1** Control descriptions are really helpful, but only if you remember to type them in! Also note the availability of a tip strip.

tool on the control's terminal on the block diagram. This is very handy when you are starting to wire up a very complex VI with many front panel items. The description should contain the following general information, where applicable:

- Purpose or function of the control or indicator
- Limits on numeric values (We usually write *Range: 0–128.*)
- Default values
- Special interactions with other control settings

*This is the most underused feature of LabVIEW.* When confronted with a new VI, a user typically has no alternative but to guess the function of each control and indicator. It is a welcome sight to have a brief message show up in the LabVIEW Help window just by pointing at the object. Get in the habit of entering a description as soon as you create the object. Then if you copy the object to other VIs, the description follows along. It's much easier than saving the documentation for later. Remember to tell users about this feature.

## Custom Online Help

LabVIEW has an online help system that replaces most of the manuals. You can add your own reference documents to this help system as HTML or as compiled help files (\*.chm). The documents could describe the usage of individual VIs or serve as a user's manual. To use compiled help files, you need to create a document (it can contain both graphics and text) and then use a specialized help compiler for your platform to generate a source file in Windows Help format.

Once you have a help document compiled, you link it to a particular VI through the Documentation section of the VI Properties menu. Particular keywords (Help Tags) in a help file can be linked as well. That makes it easy for the user to search for a topic or VI name and to jump directly to the proper part of your spiffy online help document. Developers are encouraged to place their help files in the Help directory and to use the <helpdir>: pseudopath in VIs that reference help in this directory.

When you enter a path in VI Setup's Help Path, use the current platform's path conventions. As with other paths in LabVIEW, the path will be formatted correctly when you move to another platform.

Additionally, you can place VIs and LLBs in the Help directory, and top-level VIs will be automatically listed in the Help menu. When the user selects a VI from the menu, it will be loaded (and if you configure VI Setup to execute upon load, it will start executing immediately).

You can programmatically access the online help system through the **Control Online Help** function from the Programming >> Dialog and User Interface >> Help function palette. This function lets you open and close the online help system, display the main Contents menu, or search for a specific keyword or HTML file. It's a way to make a Help button on a LabVIEW panel do something more than display a dialog box.

## Documenting the Diagram

### CLAD

While the overall VI description provides adequate coverage of the purpose and function of a VI, it leaves something to be desired when it comes to explaining the diagram. There are many helpful items that you can add to a diagram to make it easier to understand (assuming that you have laid it out in a neat and orderly fashion). Many of these techniques appear in the examples in this book.

- Label every loop and every frame of Case and Sequence structures. Place a free label (using the text tool) inside each frame that states the objective of that frame, for example, "Read data from all configured channels." Sometimes it helps to use a larger font or set the style to bold to make the label stand out.
- Label important wires, especially when the wire is very long or its source is somehow obscured. Make the foreground color transparent (T) and the background color white; then place the label right on the wire.
- Use large, boldface key numbers located near important parts of the diagram; then use a scrolling diagram string constant (located off to the side or bottom of the diagram) to explain each key item. This technique reduces clutter within the diagram but provides lots of information nearby.
- Add an index to Sequence structures. If you have a sequence with more than two or three frames, create a list outside the structure that helps the user quickly locate the desired frame.

## VI History

During development, you can track changes to VIs through the History Window, which you display by selecting the **VI Revision History** item from the Edit menu. In the History Window, you can enter descriptions of important changes to the VI and then add your comments to the history log. Revision numbers are kept for each VI, and you can reset the revision number (which also erases all history entries) by clicking the **Reset** button in the History Window.

Some kinds of information can be automatically added to this historical record. From the LabVIEW Options dialog, you have access to a number of settings associated with History logging. For instance, you can have LabVIEW automatically add an entry every time the VI is saved or prompt the user for an entry at that time. Settings made through the Options dialog are the default settings for new VIs. You can also customize the History settings for individual VIs through the VI Properties menu.

History information can be printed though the **Print** command in the File menu. In the Base and Full editions, this only prints the history for a single VI. In the Professional Edition, the Print command extracts history information for all the VIs in a hierarchy or directory.

This capability is included with LabVIEW as a tool for formal software development. There is great demand from professional LabVIEW developers for greater capability in such areas.

## Other Ways to Document

One foolproof way of getting important instructions to the user is to place a block of text right on the front panel, where it can't be missed. Like a pilot's checklist, a concise enumerated list of important steps can be invaluable. You might even put a suggestion there that says, "Select Get Info from the File menu for instructions."

If a front panel has a large string indicator, have it do double duty by putting instructions in it, then *Make Current Value Default*. Note that you can type into an indicator as long as the VI is in edit mode.

## Printing LabVIEW Panels and Diagrams

We don't believe in printing LabVIEW panels and diagrams for the purpose of basic documentation—and we're not alone in our position. The interactive, layered, hierarchical nature of LabVIEW is at best poorly represented on paper. "But my Quality Assurance manager requires printouts," you say. Well, ours did, too, but we talked him into accepting electronic files instead. What use is a big, unreadable stack of paper anyway? The point is this: *The LabVIEW paradigm is completely different from procedural languages and should not be forced into that same old mold*. Nobody is going to stop you from printing out your code, but please think before you kill another tree.

Despite our pleading, you may still have to print a few panels or diagrams. Perhaps you need to show someone your work where there is no computer running LabVIEW. Or maybe you're writing a paper (or a book!) about your LabVIEW project and you need images of various screens. In those cases, you can and should print those images. Here are some of the methods.

The **Print Window** command (from the File menu) does just that: Whichever LabVIEW window is on top gets sent to the printer, using the current page setup. Limited customization of the printout is available through **Print Options** in the VI Properties menu. In particular, you can request *scale-to-fit*, which really helps if the panel is large. When you print a front panel, note that *all* items on the panel will be printed, including those you have scrolled out of the usual viewing area. To avoid printing those items, you should hide them (choose **Hide Control** by popping up on the control's terminal).

The **Print** command is quite flexible, depending on the edition of LabVIEW that you have. You can choose which parts of the VI you wish to print (panel, diagram, connector pane, description, etc.) through the feature-laden Print dialog box. Printouts are autoscaled (if desired), which means that reasonably large panels and diagrams will fit on a single sheet of paper.

#### Putting LabVIEW screen images into other documents

There are several ways to electronically place an image of a LabVIEW screen in another application:

- Copy and paste items directly.
- Use a screen-capture utility.
- Print to rich text format (RTF) files with or without embedded images (as BMP files).
- Print to HTML files with PNG, JPEG, or GIF images.
- Print to encapsulated PostScript (EPS) files.

The big decision is whether you care if the objects are bitmaps or fully editable vector graphic objects. Bitmaps generally take up more memory and may have fairly low resolution (say, 72 dots per inch), but are generally resistant to corruption when printed. Vector graphics (such as PICT, EPS, and EMF) are more versatile because you can edit most of the objects in the image, but they sometimes fly apart when displayed or printed in certain applications.

On the Macintosh and Windows, you can select objects from a LabVIEW panel or diagram, copy them to the clipboard, and then paste them into any application that displays PICT (Macintosh) or EMF (Windows) objects, including word processors. If you paste them into a drawing application, such as Canvas or CorelDraw, the objects are accurately colored and fully editable. Diagrams look pretty good, though you may lose some wire textures, such as the zigzag pattern for

strings and the dotted lines for booleans. This is surely the easiest way to obtain a quality image from LabVIEW.

Screen captures are another quick way to obtain a bitmap image of anything on your computer's screen, including LabVIEW panels and diagrams. Both Macintosh and Windows machines have a built-in screen-capture capability. On the Macintosh, use command-shift-3, which saves a bitmap image of the entire screen to a file on the start-up disk, or command-control-shift-4 to obtain a cursor that you can use to lasso a portion of the screen and copy it to the clipboard. On Windows, the Print Screen key copies the complete screen image to the clipboard, and ALT-Print Screen copies the top window (the one you clicked last) to the clipboard. Alternatively, you can obtain one of several screen-capture utilities that can grab a selected area of the screen and save it to the clipboard or to a file. They're available as commercial packages or shareware. The nice thing about many of these utilities is that you can capture pulled menus and cursors, which is really helpful for user documentation.

LabVIEW's Print command can produce a variety of panel and diagram image types, along with all the other documentation text. If you print to an RTF file, it's easy to load into word processors. In RTF mode, you can also ask LabVIEW to save the images external to the main RTF file (they come out as a set of BMP files). Printing to HTML gives you a browser-friendly form of documentation. Images are external since HTML is intrinsically a text-only language, and you can choose from JPEG, PNG, and GIF for the image files. Keep in mind that you can use the VI Server features (as we already discussed in the Document Directory VI) to automatically generate image files, albeit with fewer choices in content and file type.

What if you're creating a really formal document or presentation and you want the very finest-quality images of your LabVIEW screens?

Here's Gary's recipe, using the PostScript printing feature of LabVIEW. It works on any Macintosh and Windows systems with certain printer drivers, such as the one for the HP LaserJet 4m; check your driver to see if it has an EPS file generation feature. You must have LabVIEW 4.01 or later and Adobe Illustrator 6.0 or later to edit the resulting images. Here are the steps:

1. Use LabVIEW's Print command with a custom setup to print whatever is desired. Your life will be simpler if you use the **scale to fit** option to get everything on one page. Otherwise, objects will be clipped or lost since EPS records only one page's worth of information.
2. In the print dialog, print to a file. Macintosh lets you choose whether the file is pure PostScript or EPS and whether it has a screen preview. Make the file type EPS with a preview.

3. Open the resulting file with Illustrator 6.0 or later (prior versions will fail to properly parse the EPS file). The panel and diagram are fully editable, color PostScript objects. You will need to ungroup and delete some of the surrounding boxes. Also, many items are masked and use PostScript patterns. Be careful what you change or delete.
4. Save the file as Illustrator EPS for importing into a page layout program. EPS files are absolutely, positively the most robust graphics files when you're dealing with complex compositions and you're nit-picky about the outcome. *Note:* If you send the file to a service bureau, make sure you take along any strange fonts that you might have chosen in LabVIEW, or have Illustrator embed them in the file.

While it takes several steps, the results are gorgeous when printed at high resolution, such as 1200 dots per inch. This book is an example of the results.

## Writing Formal Documents

If you need to prepare a formal document describing your LabVIEW project, try to follow the conventions described here. Our objective here is to help propagate a consistent style of documentation so users can more quickly understand what you have written. If you work for a big company, it may have a style guide for technical writers that should be consulted as well. General document style is beyond the scope of this book; only the specifics of content and format as they pertain to VI descriptions will be discussed.

### Document outline

For an instrument driver or similar package, a document might consist of the following basic elements:

1. Cover page
2. Table of contents
3. Introduction—“About This Package”
4. Programming notes—help the user apply the lower-level function VIs
5. Using the demonstration VI
6. Detailed description of each lower-level function VI

The first five items are general in nature. Use your best judgment as to what they should contain. The last item, the detailed description of a particular VI, is the subject of the rest of this section. For major

applications, write a user's manual and a programmer's manual. In the user's manual, begin with a *quick start* section. Limit this to only a few pages because most users have the attention span of a 2-year-old. In that section, put in annotated screen shots that describe the important controls and explain how to start up and shut down the system. Next, go into detail about the various LabVIEW panels and modes of operation. In general, you should model your manuals after those of other commercial packages that you judge to be effective.

### Connector pane picture

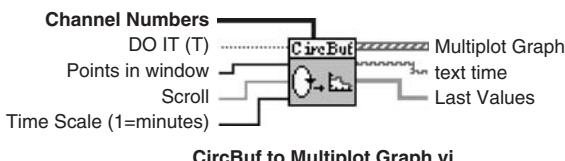
Use the methods described under “Putting LabVIEW Screen Images into Other Documents” to import an image of the connector pane into your word processor, as shown in Figure 9.2. This gives the reader a handy key to the VI’s connections, just as you find in the LabVIEW manuals.

Note the different styles of text in the figure. This is achieved by setting the **required connections** on the VI’s connector pane. While editing a VI and showing the connector pane, you pop up on a terminal point and select **This Connection Is**. The choices available are Required, Recommended, or Optional. Required terminals will automatically be displayed in the Help window (and in printouts) in boldface.

These terminals must be wired when the VI is placed on a diagram, or the calling VI will be broken. Recommended terminals (the default setting) are displayed in normal typeface. Optional terminals are those that the user will rarely have to access, and they are grayed out. In the Context Help window, the user can select **simple diagram help** or **detailed diagram help**. In the simple mode, optional terminals are not shown. This simplifies the life of novice users.

### VI description

If you have already entered a description of the VI in the Documentation box with the VI, open it, select all the text, and copy it to the clipboard to import it into a document in your word processor.



**Figure 9.2** The connector pane is an important part of the document for any VI. Different text styles can indicate the relative importance of each terminal.

The Print command in the Professional Edition of LabVIEW can extract all the text from a hierarchy. Just customize it to print only the documentation, and send the results to an RTF or text file. If you don't have the Professional Edition, you can use the Document Directory VI to do the same thing. Set it to Using Panel, and make the destination RTF or Text. With either of these methods, you'll also get all the control and indicator descriptions. Happy editing. . . .

Names of input and output terminals should be in boldface. If your word processor can do page layouts, the description can flow around the connector pane picture. Here is the text that goes with the connector pane shown in Figure 9.2:

A handy multichannel plotter for use with the Circular Buffer trending VI. Calls the Circular Buffer driver to return the desired quantity of data for one or more channels. Displays data as a multiplot graph and returns the last values for the specified channels. All channels share a common time base. Supports all the features of the Circular Buffer VI. To see this VI in action, run the Circular Buffer Example VI, then run this VI. Since the Circular Buffer is a kind of global variable, its data can be shared between these independent, top-level VIs.

### Terminal descriptions

Every input and output needs a description that includes the following information.

- The data type
- What it does
- Valid range (for inputs)
- Default value (for inputs)

This may be the same information that you entered in the control's description dialog box. A recommended standard for terminal labeling makes use of text style to indicate how frequently you need to change each item. When you create the VI, modify the font for each front panel item's label as follows: Use boldface for items that you will have to change often, use the normal font for items that you plan to use less often, and put square brackets [label] around the labels of items where the default value usually suffices. This method maps directly to the required connections behavior previously described. When you write the terminal descriptions into a document, put them in this prioritized order so the user will see the most-used items first. We like to start each line with a picture of the appropriate LabVIEW terminal data type, followed by the terminal's name, then its description, range, and defaults:

**132 Peak Voltage** sets the maximum peak output voltage. Note that the peak power setting for 2000 V cannot be greater than 50 W.

The possible values are as follows:

0: 16 V (Default)

1: 80 V

2: 400 V

3: 2000 V

Make a complete set of terminal icons that can be kept in a separate document, open at the same time as your project document. Just copy and paste the icons. Gary made such a document in Microsoft Word which he calls “LabVIEW Document Helpers.” It also includes preformatted document sections and styles that speed up and standardize the work. If you can’t handle all this graphical stuff, at least show the data type in text format. Here are some examples:

I32 I32 integer

[DBL] Array of DBL numerics

[[DBL]] 2D array of DBL numerics

cluster Cluster

ABC String

Try it out, and feel free to add your favorite documentation tidbits.

### Programming examples

If you are producing a commercial package, it’s a good idea to include some examples using your VIs. Show how the VI is used in a loop or in conjunction with graphs or an analysis operation. Above all, have a “try me first” example that is guaranteed to work! This will give the user greater confidence that the rest of the package is similarly functional. Put completed examples on disk where the interested user can find them and try them out. Make prominent mention of the existence of these examples in your documentation, perhaps in the *quick start* section.

### Distributing Documents

In this modern age, more and more documents are distributed in electronic form. Except for commercially marketed software packages, the user is expected to download some kind of document for personal viewing and/or printing. Text files are of course the least-common-denominator format, but they leave much to be desired: You can’t include graphics at all. Word processors and page layout programs can create nice documents, but they are not generally portable (although Microsoft

has done a good job of Mac-PC portability with Word, except for graphics items). Besides text and word processor files, there are some good formats available nowadays, thanks to the World Wide Web.

**Portable Document Format (PDF)** was created by Adobe Systems to provide cross-platform portability of elaborate documents. It based PDF on PostScript technology, which supports embedded fonts and graphics. You can buy Adobe Acrobat for almost any computer and use it to translate documents from word processors and page layout applications into this universal format. Users can obtain a free copy of Adobe Acrobat Reader from online sources (Adobe Systems has it on its Web site, [www.adobe.com](http://www.adobe.com)), or you can include it on your distribution media. Again, Acrobat Reader is available for many platforms. This is probably the best way to distribute high-quality documentation.

**HyperText Markup Language (HTML)** is the native format for many items on the World Wide Web, and all Web browsers can read files in this format—including local files on disk. The file itself is just ASCII text, but it contains a complex coding scheme that allows documents to contain not only text, but also links to other files, including graphics and sound. You can use Web authoring software, or LabVIEW automatically generates portable documentation in HTML through the Print command.

**Custom online help** is another viable way to distribute documents. By definition, all LabVIEW users have a way to view such documents. They can be searched by keyword, and of course you can include graphics. See “Custom Online Help,” or the LabVIEW user’s manual for more information on creating these files.

## Instrument Driver Basics

A LabVIEW **instrument driver** is a collection of VIs that controls a programmable instrument. Each routine handles a specific operation such as reading data, writing data, or configuring the instrument. A well-written driver makes it easy to access an instrument because it encapsulates the complex, low-level hardware setup and communications protocols. You are presented with a set of driver VIs that are easy to understand, modify, and apply to the problem at hand.

The key to writing drivers lies in a simple pronouncement: **RTM**—*Read The Manual*. We’re talking about the programming manual that goes with the instrument you’re writing the driver for and the online Application Note *Developing LabVIEW Plug and Play Instrument Drivers*. Without these references, you may end up frustrated or, at best, with a really bizarre driver. The guidelines on writing drivers, complete instrument drivers, as well as many development tools are gathered online at the Instrument Driver Network ([ni.com/idnet](http://ni.com/idnet)). National Instruments (NI) has made it exceptionally simple to create full-featured plug-and-play instrument drivers with the new **Instrument Driver Project Wizard** in LabVIEW 8. We’ll cover communication basics in this chapter and the Wizard in Chapter 11. Communicating with an instrument is more than just having a set of VIs.

### Finding Instrument Drivers

One reason that LabVIEW has been so widely accepted is because of the way it incorporates instrument drivers. Unlike in many competing applications, LabVIEW drivers are written in LabVIEW—complete with the usual panels and diagrams—instead of arriving as precompiled

black boxes that only the manufacturer can modify. Thus you can start with an existing driver and adapt it to your needs as you see fit. And you should adapt it because no off-the-shelf driver is going to match your application perfectly. Another important asset is the National Instruments online instrument driver repository at [ni.com/idnet](http://ni.com/idnet). It contains drivers for hundreds of instruments using a variety of hardware standards such as GPIB, RS-232/422, USB, even ancient VXI, and CAMAC. Each driver is fully supported by National Instruments. They are also *free*. Obtaining drivers online is easy with the **Instrument Driver Finder** in LabVIEW 8. From the LabVIEW Help menu select Help >> Find Instrument Driver.... The Instrument Driver Finder connects you to the instrument driver network at [ni.com/idnet](http://ni.com/idnet) and a collection of several thousand instrument drivers. Most of the drivers are commercial-grade software, meaning that they are thoroughly tested, fairly robust, documented, and supported. If you can't find the driver you need, you might call the manufacturer of the instrument or check its Web site. Many times the manufacturer will have an in-house development version you can use as a starting point. There is no need to reinvent the wheel!

National Instruments welcomes contributions to the instrument network. If you have written a new driver that others might be interested in using, consider submitting it. Consultants who are skilled in writing instrument drivers can join the Alliance Program and become Certified Instrument Driver Developers. If you have that entrepreneurial spirit, you could even *sell* your driver package, providing that there's a market. Software Engineering Group (SEG) was probably the first to do this with its HighwayView package that supports Allen-Bradley programmable logic controllers (PLCs). It's been a successful venture. A few manufacturers of programmable instruments also sell their drivers rather than place them in the library.

Existing instrument drivers are also one of your best resources for instrument programming examples. Whenever you need to write a new driver, look at an existing driver that is related to your new project and see how it was done. Standardization is key to software reuse. A standard instrument driver model enables engineers to swap instruments without requiring software changes, resulting in significant savings in time and money. You should design drivers for similar instruments (for instance, all oscilloscopes) that have a common interface in the software world. This means all the connector panes should be the same. *Rule: Every instrument should be a drop-in replacement for every other instrument of its genre.* This is especially important in the world of automated test systems requiring interchangeable hardware and software components.

## Driver Basics

Writing a driver can be trivial or traumatic; it depends on your programming experience, the complexity of the instrument, and the approach you take. Start by going to the Web site and reading National Instruments' *Instrument Driver Guidelines*, which will give you an overview of the preferred way to write a driver. Then spend time with your instrument and the programming manual. Figure out how everything is supposed to work. Next decide what your objectives are. Do you just need to read a single data value, or do you need to implement every command? This has implications with respect to the complexity of the project.

### Communication standards

The kinds of instruments you are most likely to use are “smart,” stand-alone devices that use one of several communication standards, notably **serial**, **GPIB**, **USB**, and **Ethernet**. Serial and GPIB are ancient by computer standards, but still a reliable and common interface on stand-alone instruments. USB 2.0 adds a high-speed data bus designed for multimedia but is perfectly suitable for message-based instruments. Ethernet-based instruments have to share the network bandwidth with everything else on the network. This can limit their effectiveness for deterministic acquisition or high-speed transfer, but makes it great for placing an instrument in a remote location, or sharing an instrument between multiple controllers. Fortunately VISA provides a common framework and application programming interface (API) for controlling instruments over all these busses. Instrument drivers properly written using VISA API calls can transparently change between serial, GPIB, USB, or Ethernet without modification.

**Serial instruments.** Popular serial communication interface standards include **RS-232C**, **RS-422A**, and **RS-485A**. These standards are defined by the Electronic Industries Alliance (EIA) and include specifications for cabling, connector pin-outs, signal levels, and timing. *Note that none of these standards say anything about the protocol or message formats.* This is a popular misconception. Protocols are covered by independent standards or, more often, by the will and whim of the instrument manufacturer. The advantages of serial communication are simplicity and low cost. RS-232, for instance, has been around for a long time, so there is plenty of inexpensive hardware available. The disadvantages are that serial systems tend to be slow (you can move only 1 bit of data at a time) and the protocols are sometimes burdensome in terms of programming.

**RS-232C** is the most common standard, supported by virtually all computer systems and most serial instruments. It uses the familiar 25-pin or 9-pin D-style connector. The standard defines functions for all the pins, although most often you will see just a few of the lines used: transmit data, receive data, and ground. The other pins are used for handshaking, with such functions as Clear to Send and Ready to Send. These functions are supported by the LabVIEW VISA serial port functions, if your instrument requires their use. The main limitation of RS-232 is that it is electrically single-ended which results in poor noise rejection. The standard also limits the maximum distance to 100 m, although this is frequently violated in practice. The top data rate is normally 115,200 Bd (baud, or bits per second), but some systems support higher speeds. It's also the most abused and ignored standard in the industry. Manufacturers take many liberties regarding signal levels, baud rate, connectors, and so forth. Not all RS-232-compliant devices are actually compatible. Beware!

**RS-422A** is similar to RS-232, but uses differential drivers and receivers, thus requiring two pairs of wires plus a ground as the minimum connection. Because of its superior noise rejection and drive capability, RS-422 is usable to 1200 m at speeds below 9600 Bd, and up to 1 MBd at shorter distances.

Other standards in this series are **RS-432A** and **RS-485A**. RS-432A is identical to RS-232C, except that the electrical characteristics are improved to support higher transmission rates over longer distances.

RS-485A is an extension of RS-422A that specifies greater drive current and supports *multidrop* systems (multiple talkers and listeners on one set of wires). Generally, your computer will need an adapter box or plug-in board to properly support these standards. National Instruments makes plug-in boards with up to 16 RS-485 ports. You can buy adapter boxes that provide electrical compatibility among all these standards from companies such as Black Box.

**Adding serial ports.** If you need more serial ports on your computer, consider a multiport plug-in board. One important feature to look for is a *16550-compatible UART*. This piece of hardware includes a 16-byte buffer to prevent data loss (the nefarious *overrun* error), which otherwise will surely occur. For any computer with PCI slots, there are several sources. Keyspan makes some excellent multiport plug-in boards for the Macintosh. National Instruments has multiport boards for PCI, PXI, and PCMCIA busses. You can also use a USB serial port adapter. Keyspan makes a two-port model that is very popular with LabVIEW users. For remote systems you might look at National Instruments' **Ethernet Device Server** for RS-232 or RS-485. Serial ports on the

Ethernet Device Server look like and act as locally attached serial ports configured with NI VISA.

**Troubleshooting serial interfaces.** There is great likelihood that you will encounter problems when setting up RS-232-style serial communications systems. Here are a few of the more common problems and solutions.

- Swapped transmit and receive lines. Although the standards clearly define which is which, manufacturers seem to take liberties. It's easy to hook things up backward the first time, and the whole system is dead. The answer is to use a *null modem* cable that swaps these critical lines. You can use a voltmeter to figure out which pins are transmit and receive. *Transmit* lines will deliver a steady voltage, on the order of 3 V for RS-232 and  $\pm 1.5$  V for RS-422 and RS-485. *Receive* lines generally stick around 0 V.
- Failure to properly connect the hardware handshaking lines. For instance, some instruments won't transmit until the **Clear to Send (CTS)** line is asserted. Study your instrument's manual and try to find out which lines need to be connected.
- Wrong speed, parity, or stop bit settings. Obviously, all parties must agree on these low-level protocol settings. The **Serial Port Init VI** (Serial Instrument I/O function palette) sets these parameters for most serial interfaces.
- Serial cable problems. When in doubt, a serial line activity indicator from Radio Shack is a very useful tool for verifying serial line connections and activity. If you do a lot of serial cable debugging, then a full RS-232 breakout box from Black Box or the equivalent will allow for rapid troubleshooting of serial cables.
- Multiple applications or devices trying to access the serial port. If the VISA Configure Serial Port function returns an error, make sure that there are no other programs running on your computer that might be using the same serial port.

**GPIB instruments.** Hewlett-Packard Corporation gets credit for inventing this popular communications and control technique back in 1965, calling it the *HP Interface Bus (HP-IB)*, a name it uses to this day. The Institute of Electrical and Electronics Engineers (IEEE) formalized it as **IEEE 488** in 1978, after which its common name **General-Purpose Interface Bus (GPIB)** was adopted. It's a powerful, flexible, and popular communications standard supported by thousands of

commercial instruments and computer systems. The most important characteristics of GPIB are as follows:

- It is a parallel, or bus-based, standard capable of transferring 1 byte (8 bits) per cycle.
- It is fairly fast, transferring up to about 800 Kbytes/s typically and 8 Mbytes/s maximum.
- Hardware takes care of timing and handshaking.
- It requires significantly more expensive and complex hardware than do serial interfaces.
- Distance is limited to 20 m unless special bus extender units are added.
- Up to 15 devices can coexist on one bus; up to 31 can coexist with a bus expander unit.

There is also a newer version of this standard, **IEEE 488.2**, established in 1987. The original standard didn't address data formats, status reporting, error handling, and so forth. The new standard does. It standardizes many of these lower-level protocol issues, simplifying your programming task. National Instruments' GPIB boards support IEEE 488.2, and the LabVIEW VISA library is based on it.

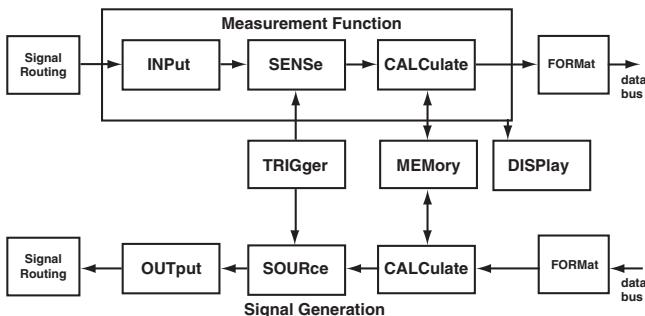
Even after the IEEE 488.2 standard was implemented, chaos reigned in the area of instrument command languages. A consortium of instrumentation companies convened in 1990 to define the **Standard Commands for Programmable Instruments (SCPI)**, and programmers' lives became much easier. SCPI defines

- Commands required by IEEE 488.2. Example: *\*IDN?* (send instrument identification).
- Commands required by SCPI. Example: *ERR?* (send next entry from instrument's error queue).
- Optional commands. Example: *DISP* (*controls selection and presentation of display information*).

Commands are structured as a hierarchical command tree. You specify a function, a subfunction, a sub-subfunction, and so on, separated by a colon (:) character. For instance, the SCPI message for an autoranging voltage measurement would be

SENSE:VOLTAGE:RANGE:AUTO

Programming a SCPI-compliant instrument can be a pleasant surprise because the command set makes sense! Also, the SCPI model



**Figure 10.1** The SCPI instrument model. Each block, or subsystem, is represented by a command hierarchy.

(Figure 10.1) covers instruments of all kinds with a general model, so the programs you write tend to be reusable. The LabVIEW Instrument Driver Wizard creates drivers based on templates that conform to the SCPI model.

## Learn about Your Instrument

You can't program it if you don't understand it. When you receive a new instrument for a driver project, you're not always familiar with its purpose or its modes of operation. Expect it to take several hours of experimenting with the controls and reading the user's manual before you're comfortable with the fundamentals. It's also a good idea to connect the instrument to some kind of signal source or output monitor. That way, you can stimulate inputs and observe outputs. It's good to have a function generator, a voltmeter, and an oscilloscope, plus test leads and accessories to accommodate most instruments. Sometimes, you need to borrow more exotic equipment. Take our advice: Don't spend too much time trying to write a driver if you don't have access to the actual instrument. Fiddle with the knobs and get acquainted. You can certainly learn the basics of the command set offline, but it's amazing how soon you will have questions that can be answered only by exchanging some messages.

Obtain and study the programming manual. Some companies write programming manuals that border on works of art, as you will find with the newer HP and Tektronix instruments. Other manuals are little more than lists of commands with terse explanations. (We'll be nice guys and let those companies remain anonymous.) Pray that you don't end up with one of the latter. By the way, programming manuals are notorious for errors and omissions. Maybe it's true because so few people actually *do* any low-level programming; they rely on people like

us to do the dirty chore of writing a driver. If you find mistakes in a manual, by all means tell the manufacturer so it can fix them in the next revision.

Even the best manuals can be kind of scary—some complex instruments have really big manuals. What you have to do is to skim through a couple of times to get a feel for the overall structure of the command set. Pay attention to the basic communications protocol, which is especially important for instruments that use the serial port. Also, figure out how the instrument responds to and reports errors. Take notes, draw pictures, and make lists of important discoveries.

Interactions between settings need special attention. For instance, changing from volts to amperes on a digital multimeter (DMM) probably implies that the range control takes on a different set of legal values. Sending an illegal range command may cause the meter to go to the nearest range, do nothing, or simply *lock up!* (Instruments are getting better about these sorts of things; in the old days, *rebooting* an instrument was sometimes required.) Your driver will need to arbitrate these control interactions, so note them as soon as they're discovered.

## Determine Which Functions to Program

Nothing is quite so overwhelming as opening the programming manual and finding out that your instrument has *4378 commands!* Obviously, you're not going to implement them all; you need to decide which commands are needed for your project.

If you are hired specifically to write an instrument driver, as we consultants are, the customer will probably supply a list of commands that you are required to support. If the list is long, you simply take more time, charge more money, and become rich. But if you are writing a driver for your own specific application, you probably don't have the time to develop a comprehensive driver package. Instead, you must determine the scope of the job and decide exactly which functions you need to implement. Here are some functions that most simple driver packages should support:

- *Basic communications.* You need the ability to write to and read from the instrument, with some form of error handling.
- *Sending commands.* You need the ability to tell the instrument to perform a function or change a setting.
- *Transferring data.* If the instrument is a measurement device such as a voltmeter or an oscilloscope, you probably need to fetch data, scale it, and present it in some useful fashion. If the instrument generates

outputs such as a waveform generator, you need to write blocks of data representing waveforms.

- *Configuration management.* You need the ability to load and store the settings of many of the instrument's important controls all at once. Some instruments have internal setup memory, while others let you read and write long strings of setup commands with ease. Still others offer no help whatsoever, making this task really difficult.
- *Important controls.* There are always a few basic controls that you simply *must* support, such as the mode and range on a DMM. More extensive drivers support more controls.

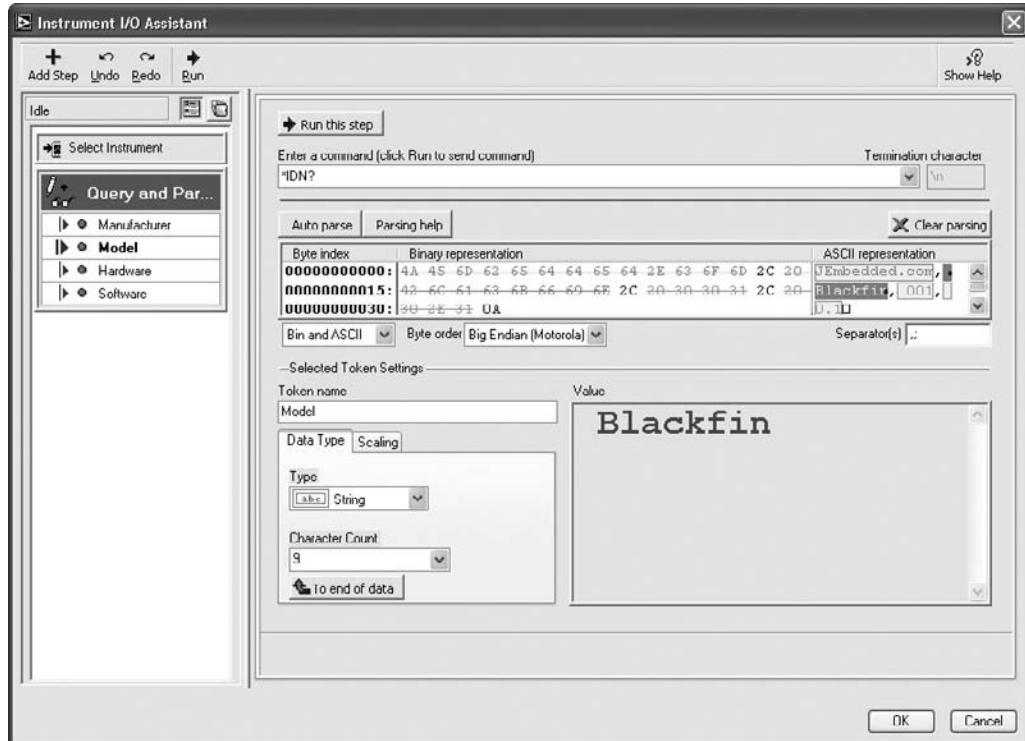
The basic question to ask is, Will the user do most of the setup manually through the instrument's front panel, or will LabVIEW have to do most of the work? If you're using LabVIEW on the test bench next to an oscilloscope, manually adjust the scope until the waveform looks OK, and then just have LabVIEW grab the data for later analysis. No software control functions are required in this case, and all you really want is a Read Data VI. If the application is in the area of automated test equipment (ATE), instrument setups have to be controlled to guarantee that they are identical from test to test. In that case, configuration management will be very important and your driver will have access to many control functions.

Think about the intended application, look through the programming manual again, and pick out the important functions. Make a checklist and get ready to talk to your instrument.

## Establish Communications

After you study the problem, the next step is to establish communications with the instrument. You may need to install communications hardware (such as a GPIB interface board); then you need to assemble the proper cables and try a few simple test commands to verify that the instrument "hears" you. If you are setting up your computer or interface hardware for the first time, consider borrowing an instrument with an available LabVIEW driver to try out. It's nice to know that there are no problems in your development system.

LabVIEW's Express VI, **Instrument I/O Assistant**, is an interactive general-purpose instrument controller that you can use to test commands one at a time. Figure 10.2 shows the “\*IDN?” query and response in the I/O Assistant. It's easy to type in a command and see how the instrument responds. The parsing window lets you sort out the reply and assign data types to data transmitted from your instrument. You can even group bytes together into multibyte data types. In Figure 10.2

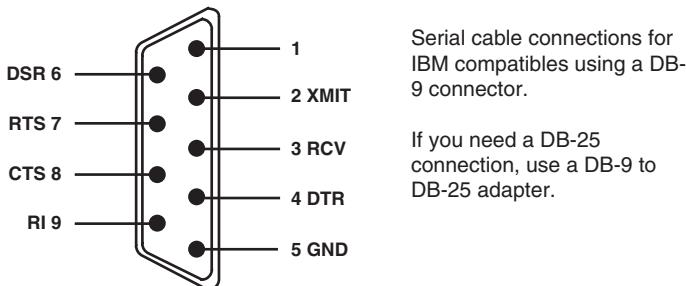


**Figure 10.2** The Instrument I/O Assistant is an interactive tool for establishing communication with instruments. The instrument in this example was built with LabVIEW Embedded for the ADI Blackfin DSP. LabVIEW not only can control instruments, but also can be the instrument firmware.

the assistant made the correct assumptions about the comma-separated data types returned by our instrument: Manufacturer (string), Model (string), Hardware Version (DBL), and Software Version (DBL). If the data types or grouping is wrong, the I/O Assistant makes it easy to select multiple bytes and group them into other data types. The I/O Assistant will even plot returned waveform data. Once you have sorted out the communication details of your instrument, the I/O Assistant will create an awful VI for you with all the commands you just tested chained together. This is exactly the kind of VI and code you do not want to use! It is not a real instrument driver. In Chapter 11 we'll look at how to build on these experimental commands and create a modular instrument driver that you can use and reuse as your application grows.

### Hardware and wiring

**Serial.** Problematic because the standards are so often violated (non-standard standards?), serial hardware can be more challenging to hook



**Figure 10.3** For PCs using a DB-9 connector, you can get cables and connectors like this at your local electronics emporium. DB-9 to DB-25 adapters are also handy.

up than the other types. If you use a lot of serial devices, a breakout box will quickly pay for itself.

Figure 10.3 shows serial port connections for the very common PC version. Cables are available with the required DB-9 on one end and a DB-9 or DB-25 on the free end to mate with most instruments. If you install a board with extra serial ports or with special interfaces (such as RS-422 or RS-485), special connectors may be required.

If you must extend the serial cable beyond a few feet, it is important to use the right kind of cable. Transmit and receive lines should reside in separate twisted, shielded pairs to prevent crosstalk (Belden 8723 or equivalent). If you don't use the correct type of cable, the capacitive coupling will yield puzzling results: Every character you transmit will appear in the receive buffer.

Your instrument's manual has to supply information about connector pin-outs and the requirements for connections to hardware handshaking lines (such as CTS and RTS). The usual wiring error is to swap transmit and receive lines on RS-232. The RS-422 adds to the problem by permitting you to swap the positive and negative lines as well. One trick you can use to sort out the wires is to test each pin with a voltmeter.

The negative lead of the meter should go to the ground pin. Transmit lines will be driven to a nice, solid voltage (such as +3 V), while receive lines will be 0 V, or floating randomly. You can also use an oscilloscope to look for bursts of data if you can force the instrument to go into *Talk* mode. A **null modem** cable or adapter usually does the trick for RS-232 signal mix-ups. It effectively swaps the transmit and receive lines and provides jumpers for CTS and RTS. Be sure to have one handy, along with some gender changers (male-to-male and female-to-female connectors), when you start plugging together serial equipment.

**GPIB.** You need some kind of IEEE 488 interface in your computer. Plug-in boards from National Instruments are the logical choice since

they are all supported by LabVIEW and they offer high performance. External interface boxes (SCSI, RS-232, USB, or Ethernet to GPIB) also use the same driver. Cabling is generally easy with GPIB because the connectors are all standard. Just make sure that you don't violate the 20-m maximum length; otherwise, you may see unexplained errors and/or outright communication failures. Incredibly, people have trouble hooking up GPIB instruments. Their main problem is failure to make sure that the connector is pushed all the way in at both ends. Always start your testing with just one instrument on the bus to avoid unexpected addressing clashes.

### Protocols and basic message passing

**GPIB.** Figure out how to set the GPIB address of your instrument, or at least find out what the address is. From the instrument's manual, determine what the command terminator is (carriage return, line feed, EOI asserted, or a combination), and pick out a simple command to try out. On the first try, you may get a time-out because the GPIB address is incorrect. If all else fails, go through all the addresses from 1 to 31 (0 is the controller) until you get a response. In really tough cases, you may have to use a GPIB bus analyzer (HP, Tektronix, National Instruments, and IOTech all make one) to examine the bus traffic. Always verify the command terminator. If the instrument expects the end or identify (EOI) signal or a special character and you don't send it, nothing will happen.

Another classic hang-up with GPIB involves the issue of **repeat addressing**. When the controller (your computer) talks to another device on the bus (an instrument), the first thing the controller has to do is to *address* the instrument. If the controller wishes to communicate with the same device again, it may take advantage of the fact that many (but not all) instruments remember that they are still addressed. If the instrument doesn't remember that it is still addressed, then you need to enable the repeat addressing action in your instrument driver. You do this by calling a VISA Property node with repeat addressing set to True.

**Serial.** Unlike GPIB, serial instruments have little in common with one another since there are essentially no standards in use. Some instruments are individually addressed so that they can be used in multidrop networks such as RS-422A. Others have no addressing and are designed for use on a dedicated link. Protocols vary from a GPIB-like ASCII message exchange to pure binary with complex handshaking. Study your instrument's programming manual and figure out what protocol it uses. The Instrument I/O Assistant can communicate using

ASCII messages or raw binary. You just need to know your instrument (*Read the Manual*).

Be sure to set the basic serial parameters: speed (baud rate), bits per character, stop bits, parity, and XON/XOFF action. LabVIEW's VISA Configure **Serial Port** function sets these parameters and more. If you use a terminal emulator, you will have to do the same thing in that program. Check these parameters very carefully. Next to swapped transmit and receive lines, setting one of these items improperly is the easiest way to fail to establish communication.

Instruments that use those nasty binary protocols with handshaking are nearly impossible to test by hand, but the I/O Assistant gives you at least a fighting chance. For these instruments, try to muddle through at least one command just to find out if the link is properly connected; then plunge right in and write the LabVIEW code to handle the protocol. Hopefully, you will be 90 percent successful on the first try because debugging is difficult.

As with GPIB, you can use a serial line analyzer (made by HP, Tektronix, and National Instruments) to eavesdrop on the communications process. The analyzer stores lots of characters and decodes them to whatever format you choose (ASCII, hexadecimal, octal, etc.). This really helps when your driver *almost* works but still has reliability problems.

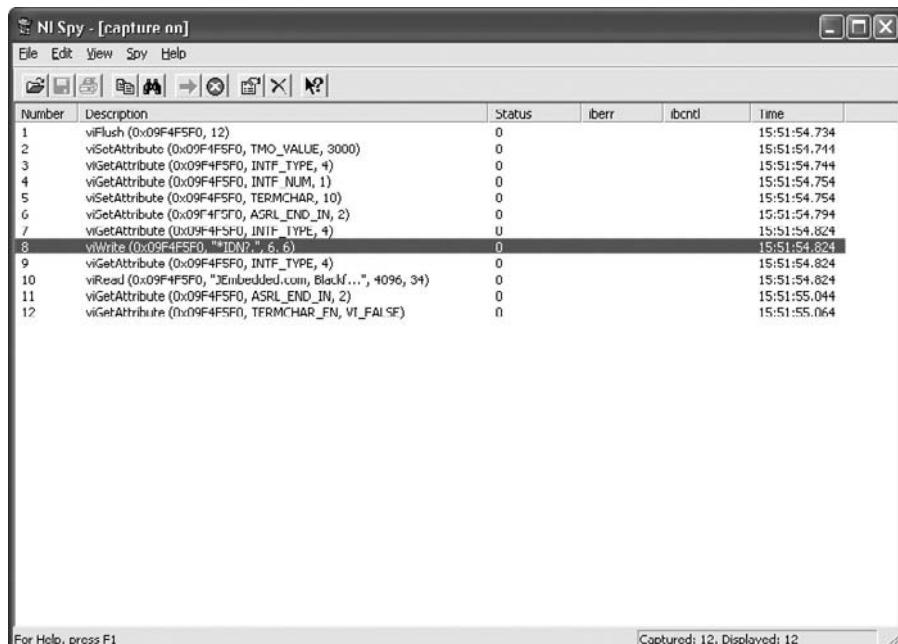


Figure 10.4 NI Spy eavesdrops on communications as a troubleshooting aid.

If you just need to eavesdrop on the communication but don't need to monitor each status line, then NI Spy is what you need. NI Spy can log every VISA call from any application that communicates using VISA. Figure 10.4 shows a capture file from NI Spy.

In Chapter 11, we'll start writing drivers by using a modern methodical approach.

## Bibliography

**1999 SCPI Syntax & Style**, SCPI Consortium 2515 Camino del Rio South, Suite 340 San Diego, CA, 1999

# Instrument Driver Development Techniques

With every major release of LabVIEW, we've watched the general architecture of instrument drivers change significantly. Back in the time of LabVIEW 1, about the only I/O functions we had were GPIB read and write, and serial port read and write. Because the product was so new, driver architectures weren't really defined, and there was a degree of chaos in the blossoming driver library. With LabVIEW 2, we gained IEEE-488.2 functions and the important application note *Writing a LabVIEW 2 Instrument Driver*, which gave developers directions regarding hierarchical program design. In the time of LabVIEW 3, those of us involved in driver (and DAQ) development standardized on **error I/O** as a common thread. It really cleaned up our diagrams, eliminated many Sequence structures, and unified error handling. VME Extensions for Instrumentation (VXI) instruments appeared, as did the initial **VISA Transition Library (VTL)**. (VTL was implemented as a wrapper around the existing GPIB functions, rather than as native functions.) Many consultants launched their careers by writing drivers for VXI instruments, some of them using VTL as the foundation. Since LabVIEW 4, VISA is the primary platform for driver development—in LabVIEW and in many other instrumentation control languages.

The **Virtual Instrument Software Architecture (VISA)** provides a standard communication driver to handle all forms of instrument I/O, whether the bus is GPIB, VXI, PXI, Serial, Ethernet, or USB. VISA is an accepted standard and an integral part of LabVIEW. The principle is simple: There is a consistent interface to all instruments based on the concept of Open, Read/Write, and Close, much like file operations. (See Figure 11.1.) When an instrument is opened, a **VISA session**

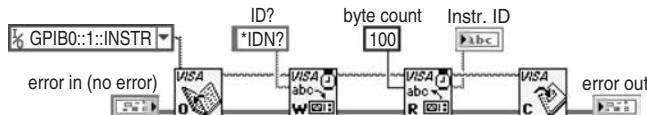


Figure 11.1 VISA Open, Write, Read, Close functions.

identifier (like a file refnum) is created to uniquely identify that instrument. Data is exchanged in string format for message-based instruments or as words or bytes for register-level access. Events, such as triggers and service requests, are a part of the standard, as is error handling. Today LabVIEW's GPIB and serial routines use VISA underneath—whether you knew it or not, if you talk to an instrument with LabVIEW, you're using VISA. The major advantage, of course, is that any driver you write with VISA is immediately portable across all VISA-supported computer platforms and busses without any additional programming on your part.

Just as VISA provides a standard communication driver, **Standard Commands for Programmable Instruments (SCPI)** defines a common set of programming commands. SCPI works by describing manufacturer agreed upon commands and command style across all instruments. SCPI defines the messages you send to an instrument, how the instrument replies, and how transmitted data is formatted. Figure 11.2 shows the SCPI commands used to get voltage (MEAS:VOLT?) and current (MEAS:CURR?) from a power supply. The reasoning behind SCPI is simple: If two instruments have the same controls, the same measurement capabilities, and the same basic functions, then it makes sense that you should be able to use the same commands. Of course, this is the real world, and every manufacturer thinks its unique instrument requires unique commands. But at least you have a fighting chance if they follow a SCPI format. Learning a completely new command set and data format for every new instrument is a pain!

Instrument drivers have come a long way since LabVIEW 4; now VISA and SCPI are the standard for all LabVIEW instrument drivers. But just because we're all using the same tools doesn't mean the drivers will work well together, or, better yet, be interchangeable. To create

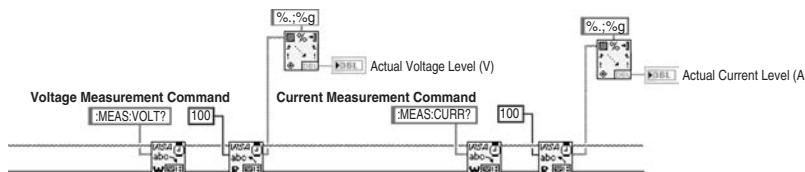


Figure 11.2 Standard Commands for Programmable Instruments simplifies instrument commands.

instrument drivers that truly work well together, we need to use the same templates and follow formal guidelines. After all, you must have standards, recommended practices, and quality assurance mechanisms in place if you expect to produce reliable, professional-grade software.

## Plug-and-Play Instrument Drivers

LabVIEW **Plug-and-Play instrument drivers** have a common look and feel, so you can operate the instrument with a minimum of programming. Additionally, each plug-and-play instrument driver is interchangeable with drivers of similar instruments. Creating new plug-and-play instrument drivers sounds like a tall order, and it indeed would be, were it not for new tools based on the guidelines and product suggestions of the **LabVIEW Instrument Driver Advisory Board**. The board is made up of leading LabVIEW users who help set standards and recommend improvements to LabVIEW. Many of their suggestions have been incorporated into the LabVIEW 8 **Instrument Driver Project Wizard**. The new Instrument Driver Project Wizard provides templates, documentation tools, and icon creation tools for the most common instrument types. This new tool really makes it easy to create Plug-and-Play instrument drivers. You don't even have to be a LabVIEW guru. (See Figure 11.3.)

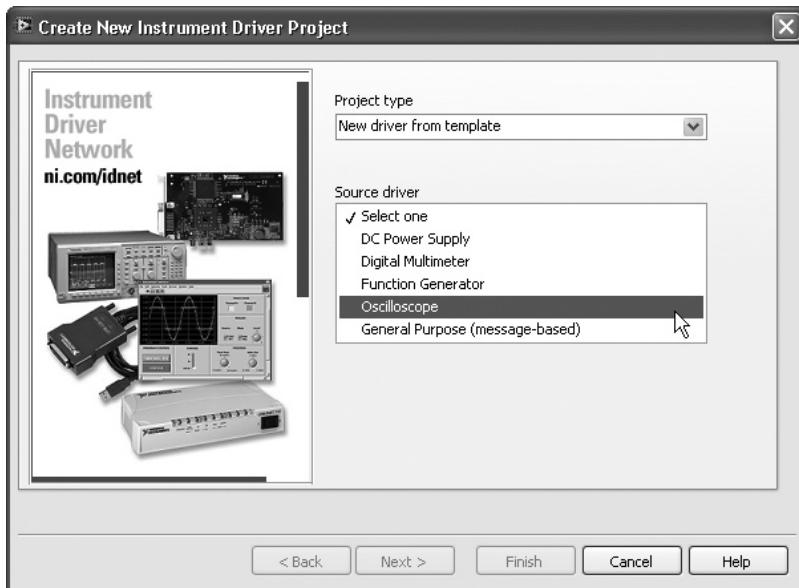


Figure 11.3 The Instrument Driver Project Wizard has templates for common instrument types.

But don't rely totally on the wizard; it's a great framework to start with, but properly designed instrument drivers need to conform to the guidelines found at the National Instruments(NI) Web site [www.ni.com/idnet](http://www.ni.com/idnet). The two most important documents that you *must* read before proceeding with driver programming are as follows:

- *Developing LabVIEW Plug-and-Play Instrument Drivers* "describes the standards and practices for LabVIEW Plug-and Play-drivers, is intended for driver developers and describes the standards for structuring VIs, instrument I/O, style, and error reporting. It also describes instrument driver components and how to integrate these components. In addition, this document describes a process for developing useful instrument drivers."
- *Instrument Driver Guidelines (checklist)* "should be used to verify that your instrument driver complies with the Instrument Driver Library standards. These guidelines provide the requirements and recommendations for programming style, error handling, front panels, block diagrams, icons, testing, and documentation, and should be referenced throughout the driver development process. Use these guidelines as a checklist to review your driver for instrument driver certification."

To help you get started in the right direction, we've incorporated NI's recommendations and provided a few examples using the Instrument Driver Project Wizard that represent good driver design techniques. Even if you're writing a driver only for yourself, it's worth following these guidelines because they will help you structure your drivers and, in the end, become a better programmer.

## General Driver Architectural Concepts

Building an instrument driver is no different from any other LabVIEW project. You still need to define the problem, do some preliminary testing and prototyping, and then follow good design and programming practices. Follow up with thorough testing and documentation, and you will have a versatile product that is easy to use and easy to maintain. The LabVIEW project created by the Instrument Driver Project Wizard uses driver template VIs to create a project complete with core VIs, use case examples, and documentation. Once you've created the project, it's easy to go through the VIs and modify the commands to match the format required by your instrument. As you go through the process, edit the documentation; and when you're done, you will have a professional-quality instrument driver.

## Error I/O flow control

An important characteristic of a robust driver is that it handles error conditions well. Any time you connect your computer to the outside world, unexpected conditions are sure to generate errors, and the instrument driver software is the place to trap those errors. Your response depends on the nature and severity of the error, the state of the overall system, and options for recovery.

The most straightforward error response is a **dialog box**, but avoid placing dialog routines down inside the driver layer. The last thing an unsuspecting developer using your driver wants is a surprise dialog box halting a running system. Dialogs are best placed at a higher application layer and not in the driver. Assuming that an operator is always available to respond, the dialog box can present a clear statement about the error: who, what, when, where, and why. Dialogs are always appropriate for fundamental errors such as basic setup conflicts (“No GPIB board in slot 3”) and for catastrophic errors where the program simply cannot continue without operator intervention (“The power supply reported a self-test failure and has shut down”). It’s also helpful to receive informative dialogs when you are setting up an instrument for the first time. As you develop your driver, make sure you squash every bug and answer every mysterious error as they pop up during development. You should never ship a driver that generates unexplained errors.

Automatic error recovery is used often in serial communications systems, particularly those that span great distances and/or run at high speeds. Local-area networks typically have sophisticated means of error detection and automatic message retransmission. For a serial instrument, you may be able to attempt retransmission of data if a time-out occurs or if a message is garbled. Retransmission can be added to a state machine-based driver without too much trouble. If a message is invalid, or if a time-out occurs, you could jump back to the first frame of the sequence and send the command again, hoping for success on the second try. An important feature to include there would be a limit on the number of retransmissions; otherwise, you would end up in an infinite loop if the instrument never responded. Such automatic actions should be limited to the lowest levels of a driver package, as in the example of a serial interface. Serious measurement and control applications are designed to report *any* error that is detected, to guarantee data quality and system reliability. Therefore, you should think carefully before making your program so smart that it “fixes” or ignores potentially dangerous error conditions.

You can use error handling to promote dataflow programming and simplify applications of a driver package. An error I/O cluster containing

error information connects one VI to the next. If an error is generated by one VI, the next one will be notified of the error and be able to take appropriate action. For instance, if a function that initializes the instrument fails, it probably makes no sense to continue sending commands. The error I/O chaining scheme can manage this decision at the driver level, rather than forcing the user to add Case structures to the top-level VI. VIs respond to an incoming error by skipping all activity and simply passing the incoming error through. Decisions about how to handle the error are made by the application developer at a higher level.

A canonical VI with error I/O is shown in Figure 11.4. The **error in** and **error out** clusters always appear in the connector pane as shown. Doing so permits all users of error I/O to line up their VIs in an even row. These clusters are available as standard controls in the Array and Cluster control menu. The cluster contents are always the same:

**Item 1: Status** (boolean). *True* means an error occurred. *False* implies a warning if the error code number is nonzero.

**Item 2: Code** (I32). An error code number used for message generation.

**Item 3: Source** (string). Contains the name of the VI calling chain that generated the error.

On the diagram, the **error in** boolean is tested, and if it's True, you may run a special error response program or simply pass the error along to the **error out** indicator and do nothing else. If no incoming error is detected, then it's up to your program to generate an error cluster

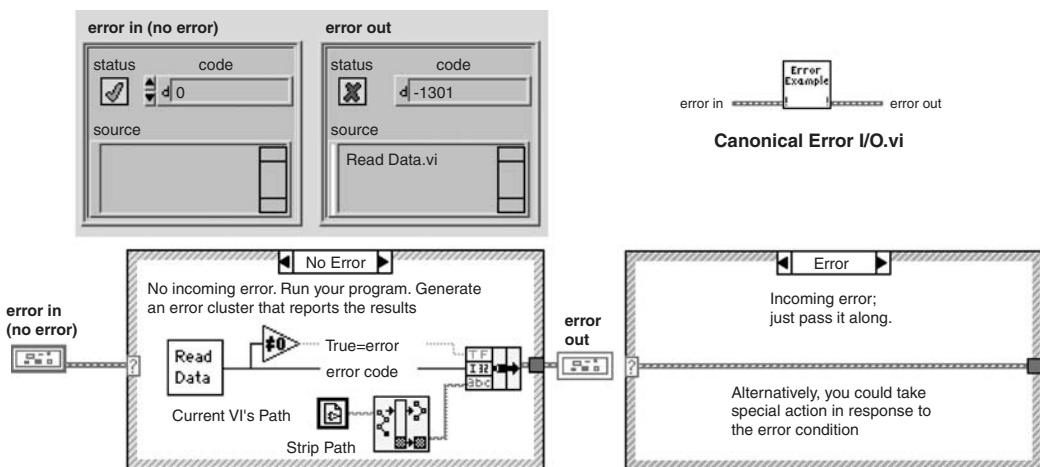
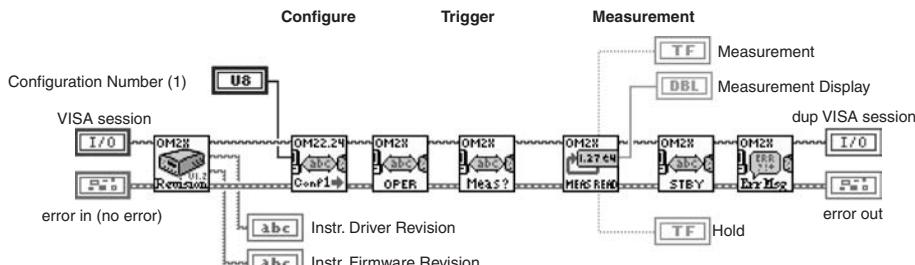


Figure 11.4 The Canonical error I/O VI. You can encapsulate any program inside the False case.



**Figure 11.5** An example of error in/error out, the AOIP Instrumentation OM22 microohmmeter, a GPIB instrument. Note how clear and simple this application is, with no need for flow control structures. Error I/O is an integral part of the underlying VISA driver.

of its own, depending on the outcome of the operation that it performs. Figure 11.4 shows a couple of tricks in case your subVI doesn't generate an error cluster. Generate the **Status** boolean with the help of the Not Equal comparison function. Enter the name of the VI into the **Source** string automatically by using the Current VI's Path constant and the Strip Path function, both from the File I/O function menu.

Figure 11.5 is an example of error I/O in action in the driver for a GPIB instrument. Error I/O is an integral part of all VISA drivers. This technique makes it easy to do your error handling in a consistent fashion all the way from the bottom to the top of the hierarchy. As you can see from the diagram, applying the AOIP Instrumentation OM22 driver is very easy because of the use of error I/O.

The VISA functions have error I/O built into the function. Each VISA function checks the error status boolean to see if an error occurred upstream. If there is an error, the VISA function does nothing and passes the error message along without modification through the error out indicator. If there is not an error, the VISA function acts normally. You can take advantage of this inherent error handling to clean up and simplify your diagrams. You also want to be alert to any errors and correct them as soon as they occur.

At some point in your application, usually at the end of a chain of I/O operations, the error cluster is submitted to an error handler. The error handler decodes the *code* number into plain language and appends the *source* string. That way, the user will understand what the error is and where it came from without resorting to an error message table or a lot of debugging. The message is generally displayed in a dialog box.

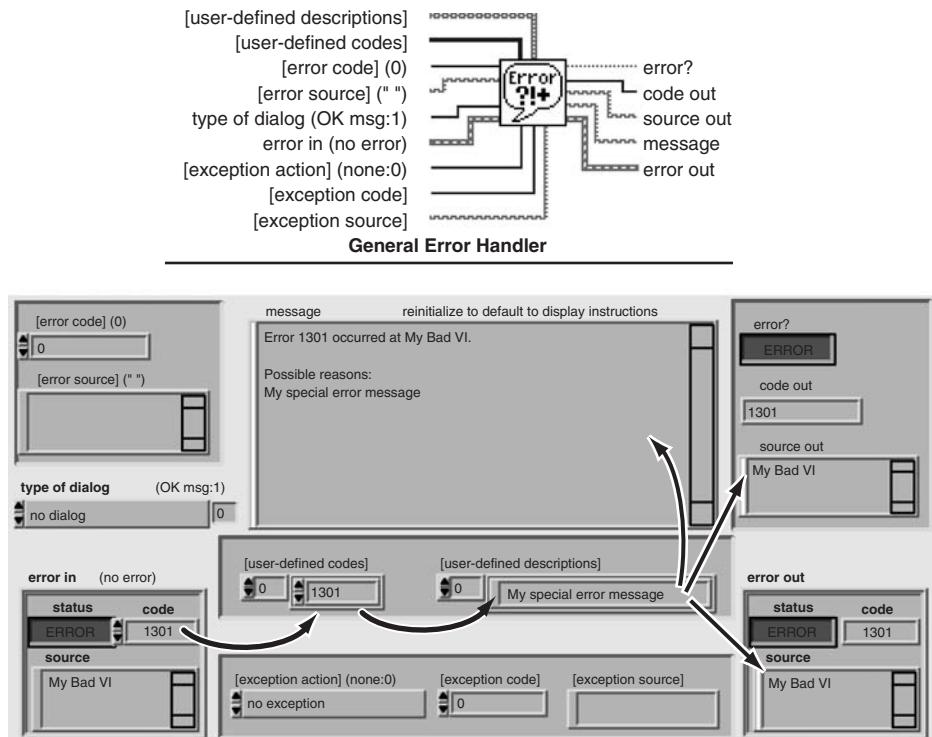
Error codes are somewhat standardized. Zero generally means no error. Positive numbers are warnings, and negative numbers are errors. Table 11.1 lists some of the error codes that are reserved for use with generic instrument driver VIs; there are many more codes defined by the VISA driver as well. If you use one of these codes in your own

**TABLE 11.1 Predefined VISA Error Codes for Instrument Drivers**

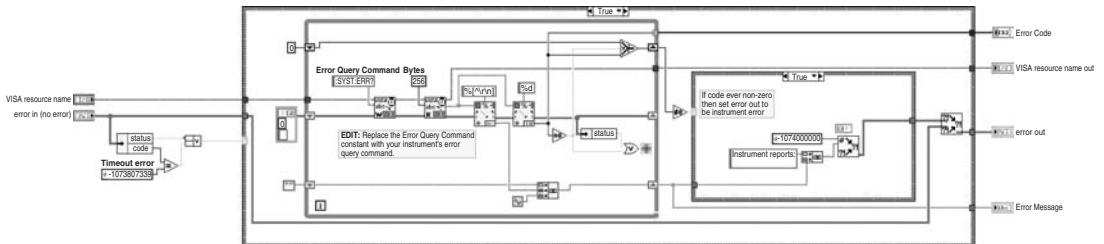
Code	Description
0	No Error: the call was successful
1073481728 to 1073483775	Warning: Developer defined warnings
-1074003951	Error: Identification query failed
-1074000000	Error: Instrument Defined Error
-1073999873 to -1074001919	Error: Developer defined errors

drivers, the General Error Handler VI that comes with LabVIEW will be able to interpret them. (Codes from 5000 to 9999 are reserved for miscellaneous user-defined errors; they are not automatically interpreted by the error handlers.)

The **General Error Handler** utility VI (Figure 11.6) accepts the standard error cluster or the individual items from an error cluster and decodes the code numbers into a message that it can display in an error dialog and in an output string. A useful feature of this handler is its



**Figure 11.6** The General Error Handler utility VI is all you need to decode and display errors. The arrows indicate how the VI can decode user-defined error codes and messages that you supply, in addition to the predefined error codes.



**Figure 11.7** The Error Query VI reads the instrument’s error buffer. Run this VI after each command or set of commands to get fresh error messages.

ability to decode user-defined error codes. You supply two arrays: *user-defined codes* and *user-defined descriptions*. The arrays have a one-to-one correspondence between code numbers and messages. With this handler, you can automatically decode both predefined errors (such as GPIB and VXI) and your own special cases. We would suggest using the codes in Table 11.1 because they are reserved for instrument-specific errors.

Error I/O simplifies diagrams by eliminating the need for many flow control structures. By all means, use it in all your drivers and other projects.

If you are programming a SCPI-compliant instrument, you can read errors from the standardized SCPI error queue. A driver support VI, **Error Query**, polls the instrument and returns a string containing any error messages via the regular error out cluster. (See Figure 11.7.) Add this VI after every command or series of commands. Events are stored in a queue—oldest events are read out first—so you need to read events out after every command to keep from filling the queue with old, uninteresting events. This technique will add additional communications overhead in the event of an error, but it will catch every error as it happens.

### Modularity by grouping of functions

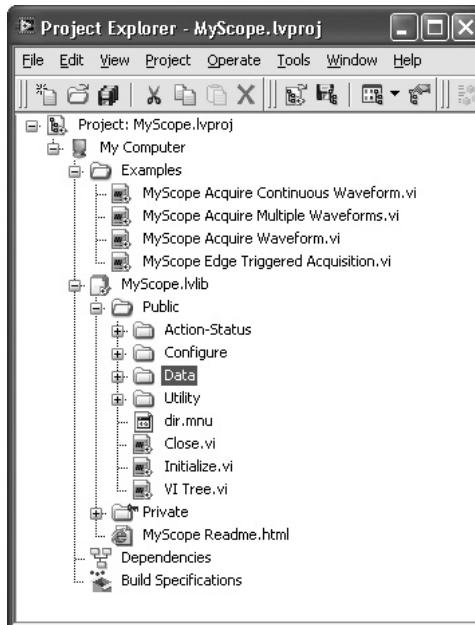
We’ve seen two extremes of **modularity** applied to LabVIEW instrument drivers: *too much* and *none*. Too much modularity exists when there is one subVI for every individual command. There are several problems with this approach. First, the interaction between control settings is really hard to arbitrate because each function has to query the instrument to see what state it’s in, rather than just comparing a few control settings on the VI’s panel. Second, the user has to plow through the directories and libraries, searching for the desired command. Third, there are just too darned many VIs, which becomes a nightmare for the poor person who has to maintain the library.

Guess what happens when you find a mistake that is common to all 352 VIs? *You get to edit every single one of them.*

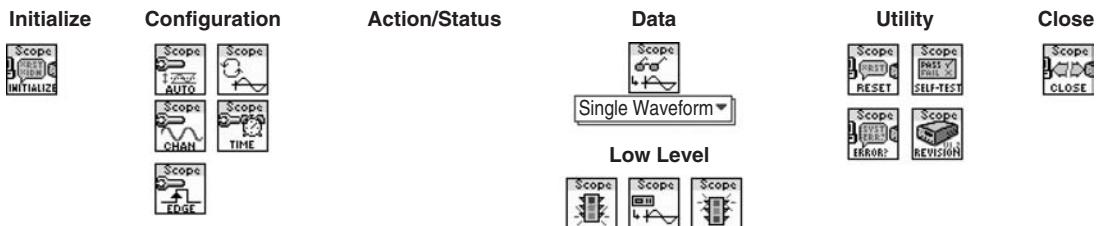
Having no modularity is the other extreme. You may find it in trivial drivers in which there are only a couple of commands and hence just one VI. You may also find a lack of modularity in complex drivers written by a novice, in which case it's a disaster. There is nothing quite like a diagram with sequences inside of sequences inside of sequences that won't even fit on a 19-in monitor. In fact, these aren't really drivers at all. They are actually dedicated applications that happen to implement the commands of a particular instrument. This kind of driver is of little use to anyone else.

### Project organization

The correct way to modularize a driver in LabVIEW is to build VIs that group the various commands by function. In the LabVIEW Plug-and-Play Instrument Driver model, VIs are organized into six categories: Initialize, Configuration, Action/Status, Data, Utility, and Close. (See Figure 11.8.) These intuitive groupings make an instrument easier to learn and program. The wizard automatically creates stub VIs for you



**Figure 11.8** The Instrument Driver Project Wizard automatically creates a project organized with core VI stubs, examples, and documentation.



**Figure 11.9** Default VI Tree for oscilloscopes. Common commands, grouped by function, are easy to find and use. You create Action/Status VIs as needed.

that implement the basic commands for each instrument. The project is ordered and grouped according to function, and a VI Tree is created to make the interrelationship easy to see. The default VI Tree for oscilloscopes is shown in Figure 11.9. Once you've created the project by using the wizard, you will need to modify the specific commands inside each VI to match your instrument. Any additional functions should be created as new VIs in the project hierarchy and placed on the VI Tree.VI. The one thing you never want to do is to change the connector pane or the functionality of the default VIs! Doing so will break their plug-and-play nature. The templates are designed to expose the common functionality of each class of instrument in a way that will make the instruments and the instrument drivers interchangeable in automated test systems. If you've developed a test system before, you can appreciate the value of replacing hardware without having to rewrite software. Because the instrument drivers are plug-and-play with the same connector pane and the same functionality, an application can be written using VI Server to dynamically call instrument driver plug-ins that initialize, configure, and take data. Which plug-in VI Server calls depends on the hardware attached to the computer. Of course, this all works seamlessly only if each of the VIs has the same connector pane and generic functionality.

### Initialization

The cleanest way to begin any conversation with an instrument is to create an **initialization VI** that starts out by calling **VISA Open** to establish a connection. The initialization VI should take care of any special VISA settings, such as serial port speed or time-outs, by calling a **VISA Property node**, and it should also perform any instrument initialization operations, such as resetting to power up defaults or clearing memory. Instruments that are SCPI-compliant can be queried with `*IDN?`, which will return an identification string that you can check to verify that you are indeed talking to the right instrument.

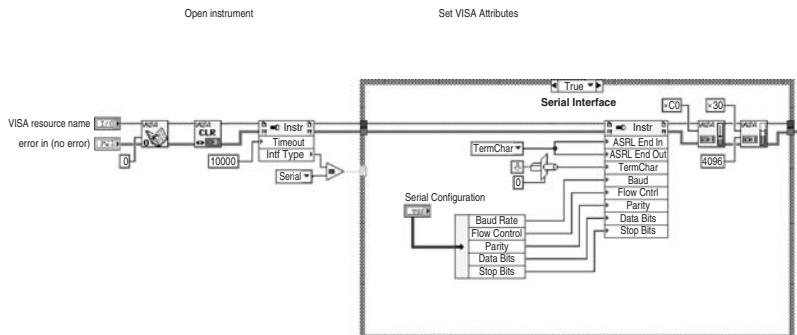


Figure 11.10 Initialize.VI opens the VISA connection and configures port speeds.

An important item to configure in your initialization VI is the **termination character**. Many instruments use a special character, such as a carriage return or line feed, to indicate the end of the message. VISA can automatically detect a specified character when receiving data (you must append it yourself when you build a string to send, however). The Initialize.VI in Figure 11.10 sets the termination character to a line feed (0x0A) by type casting the line feed constant to an unsigned byte.

## Configuration

Configuration VIs are the set of software routines that get the instrument ready to take data. They are not a set of VIs to store and save instrument configuration—those go under the utility section. A typical configuration VI, Configure Edge Trigger.VI, is shown in Figure 11.11.

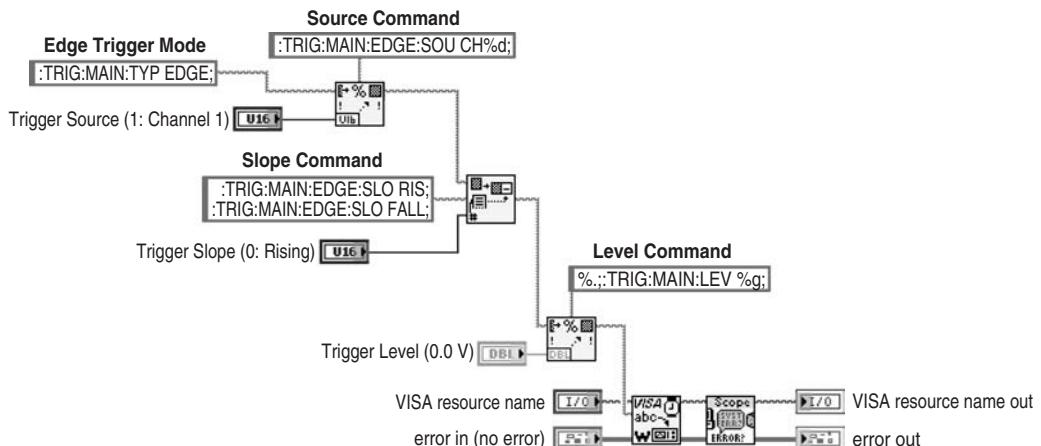


Figure 11.11 Configure Edge Trigger VI. Simple dataflow and logical control names make this block diagram easy to understand. Edit the command string for your instrument.

It concisely implements the basic capabilities of the instrument's trigger section. This VI is organized in its most basic form and is typical of all good driver VIs. Note some of its important attributes: error I/O clusters, VISA sessions as input and output common threads, and a simple dataflow diagram that builds a command string for transmission to the instrument. If a user needs to add a more advanced command, there is plenty of panel and diagram space to do so.

An important part of making your driver robust is your expectation that the user will enter unexpected values into every control. At the programmatic level, you must add some code on the diagram for range checking. The **In Range And Coerce** function and the various comparison functions are candidates for range checking. If the values are not evenly spaced (such as a 1-2-5 sequence), use a Case structure or a ring control, or select from a list. Changing the **representation** of a numeric control or indicator may also limit the range of the input, particularly for integer types. For instance, a U8 control can only represent integers between 0 and 255. Anything greater than 255 is coerced to 255, and anything less than 0 is coerced to 0.

Other difficult situations must be handled programmatically. Many instruments limit the permissible settings of one control based upon the settings of another. For example, a voltmeter might permit a range setting of 2000 V for dc, but only 1000 V for ac. If the affected controls (e.g., Range and Mode) reside in the same VI, put the interlock logic there. If one or more of the controls are not readily available, you can request the present settings from the instrument to make sure that you don't ask for an invalid combination. This may seem like a lot of work, but we've seen too many instruments that go *stark, raving bonkers* when a bad value is sent.

String controls don't have a feature analogous to the Data Range dialog; all the checking has to be done by your program. LabVIEW has several string comparison functions that are helpful. You can use **Empty String/Path?** to test for empty strings and path names, which are probably the most common out-of-range string entries. The other string comparison functions—**Decimal Digit?**, **Hex Digit?**, **Octal Digit?**, **Printable?**, **White Space?**, and **Lexical Class**—give you important attributes for the first character in a string. Being polymorphic, they also act on the ASCII equivalent of a number. For instance, if you wired the integer number 0 to the **Printable?** function, the function would return *False*, since the ASCII equivalent of 0 is *NUL*, which is not printable. Boundary conditions—values such as zero, infinity (*Inf*), not-a-number (*NaN*), and empty string—cause lots of trouble in instrument drivers. Use the comparison functions to prevent defective values from finding their way to the instrument, and always *test* your drivers to make sure that ridiculous inputs don't stop the show.

## Action and status

VIIs that start or stop an instrument's operation are action VIIs. Action VIIs arm a trigger or start the generation or acquisition of a waveform. Unlike configuration VIIs, they do not change settings, but cause the instrument to act based on its current settings. Status VIIs are used by other VIIs to obtain the current status of the instrument.

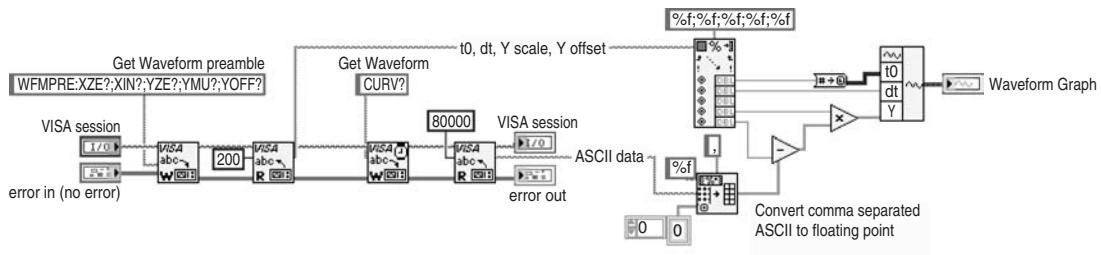
What happens when your 500-GHz 986ZX computer sends a bunch of commands in rapid-fire mode to a poor, unsuspecting instrument? Will the commands be buffered up in high-speed memory for later processing? Or will the instrument simply halt and catch fire? What if there is a communications error? Instruments are *not* infinitely fast—they require some time to respond to commands—and your driver must synchronize itself to the needs of the instrument. Timing and handshaking techniques, both hardware and software, are available and necessary to manage these situations.

Sending commands at the wrong time can be a problem. Any command that initiates a time-consuming operation has the potential for putting the instrument offline for awhile. For instance, if you tell your instrument to perform an extensive self-diagnostic test, it may ignore all GPIB activity until the test is completed, or it might buffer the commands for later execution. Also, the *rate* at which commands are sent can be important. Most instruments permit you to send many commands in one string. The question is, Will all the commands in the string be received before the first one is executed? Most times, the answer is yes, but there may be dire implications if the instrument ignores one-half of the commands you just sent. Experience indicates that testing is the only way to be sure.

There are two ways to solve these timing problems. First, you can add delays between commands by sending them one at a time in a timed Sequence structure. The second method is to find out if the previous operation is complete. If you enable **service requests (SRQs)** for your instrument, you can use **Wait For RQS**, from the VISA >> VISA Advanced >> Event-handling function palette. This technique is often used with digital oscilloscopes. Either way you should check your instrument often for errors, and report them as they happen.

## Data

Data VIIs should transfer data to or from the instrument with a minimum of user involvement. The VI in Figure 11.12 queries the scope for scaling information ( $t_0$ ,  $dt$ , Y scale, and Y offset) and then builds a waveform with the data. The VI is clean and simple, VISA and error I/O, and waveform data. In Figure 11.12 the data is returned as a comma-separated ASCII string. Sometimes transferring data as ASCII is



**Figure 11.12** ASCII data is converted to floating-point waveform.

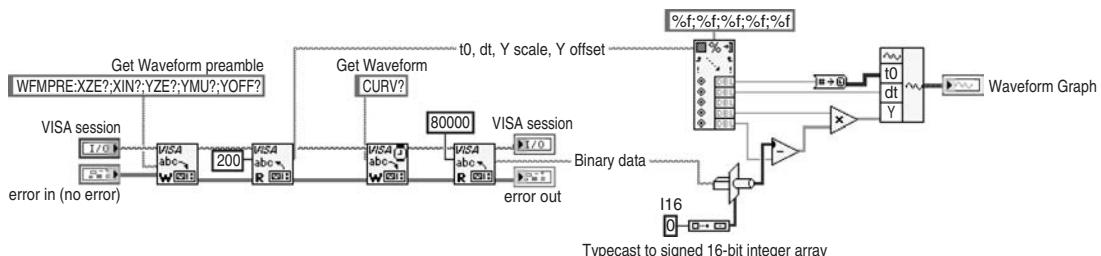
the easiest, and most instruments support it. But if you have a choice, binary protocols are faster and more compact.

The VI in Figure 11.13 is the same as that in Figure 11.12 except the data has been returned as 16-bit signed binary. You can easily change from a binary string to an array of I16 with the type cast function. Additional formatting may be required if your binary waveform has a length header or termination characters. Consult your instrument's manual for the proper handling method. LabVIEW provides all the tools you need to reconstruct your data, including functions to swap bytes and words. When you are transferring binary data, be sure your VISA Read function does not terminate on an end-of-line character. Use a VISA Property node to disable this before the read and reenable after the read; otherwise, you may not get all your data.

## Utility

Sometimes a command doesn't fit with anything else, and it is normally used by itself, for instance, utility VIs used for instrument reset, self-test, calibration, or store/recall configuration. It would be illogical to throw those VIs in with other unrelated functions.

A good driver package allows you to save and recall collections of control settings, which are usually called *instrument setups*. This saves



**Figure 11.13** Binary data is converted to floating-point waveform.

much time, considering how many controls are present on the front panels of some instruments. Sophisticated instruments usually have built-in, nonvolatile storage for several settings, and your driver should have a feature that accesses those setup memories. If the instrument has no such memory, then write a program to upload and save setups on disk for cataloging and later downloading. You may want to provide this local storage even if the instrument *has* setup memory: What if the memory fails or is accidentally overwritten?

*Historical note:* Digital storage oscilloscopes (DSOs) gradually became so complex and feature-laden that it was considered a badge of honor for a technician to obtain a usable trace in less than 5 minutes—all those buttons and lights and menus and submenus were really boggling! The solution was to provide an *auto-setup* button that got something visible on the display. Manufacturers also added some means by which to store and recall setups. All the good DSOs have these features, nowadays.

Some instruments have internal setup memory, while others let you read and write long strings of setup commands with ease. Still others offer no help whatsoever, making this task really difficult. If it won't send you one long string of commands, you may be stuck with the task of requesting each individual control setting, saving the settings, and then building valid commands to send the setup back again. This can be pretty involved; if you've done it once, you don't want to do it again. Instead, you probably will want to use the instrument's built-in setup storage. If it doesn't even have *that*, write a letter to the manufacturer and tell the company what you think of its interface software.

### Close

The Close VI is at the end of the error chain. Remember that an important characteristic of a robust driver is that it handles error conditions well. You need to make a final check with your instrument and clear out any errors that may have occurred. Error I/O is built into the VISA functions. (See Figure 11.14.) If there is an incoming error, the function will do nothing and pass the error on unmodified. The VISA Close function is the exception. VISA Close will always close the session before passing any error out.



Figure 11.14 Close VI checks for instrument errors before closing the connection.

## Documentation

One measure of quality in instrument driver software is the documentation. Drivers tend to have many obscure functions and special application requirements, all of which need some explanation. A driver that implements more than a few commands may even need a function index so that the user can find out which VI is needed to perform a desired operation. And establishing communications may not be trivial, especially for serial instruments. Good documentation is the key to happy users. Review Chapter 9, “Documentation.” It describes most of the techniques and recommended practices that you should try to use.

When you start writing the first subVI for a driver, type in control and indicator descriptions through the **Description** and **Tip** pop-up menus. This information can be displayed by showing the Help window and is the easiest way for a user to learn about the function of each control. If you enter this information right at the start, when you’re writing the VI, you probably know everything there is to know about the function, so the description will be really easy to write. You can copy text right out of the programming manual, if it’s appropriate. It pays to document as you go.

The Documentation item in the **VI Properties** dialog box from the File menu is often a user’s only source of information about a VI. The information you enter here shows up, along with the icon, in the Context Help window. Try to explain the purpose of the VI, how it works, and how it should be used. This text can also be copied and pasted into the final document.

Try to include some kind of document on disk with your driver. The issue of platform portability of documents was mentioned in Chapter 9, and with drivers, it’s almost a certainty that users of other systems will want to read your documents. As a minimum, paste all vital information into the Get Info box or a string control on a top-level example VI that is easy to find. Alternatively, create a simple text document. If you’re energetic and you want illustrations, generate a PDF (Portable Document Format) document by using Adobe Acrobat, and make sure it stays with your driver VIs; or create a LabVIEW Help document and create hyperlinks to appropriate sections in the VI Properties dialog box. The links will show up in the Context Help window.

If you’re just doing an in-house driver, think about your coworkers who will one day need to use or modify your driver when you are not available. Also, remember that the stranger who looks at your program 6 months from now may well be *you*. Include such things as how to connect the instrument, how to get it to “talk,” problems you have

discovered (and their resolutions), and of course a description of each function VI. Not only is such information useful, but also it may make you famous rather than infamous.

## Bibliography

- Developing LabVIEW Plug and Play Instrument Drivers*, [www.ni.com/idnet](http://www.ni.com/idnet), National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2006.
- Instrument Communication Handbook*, IOTech, Inc., 25971 Cannon Road, Cleveland, Ohio, 1991.
- Instrument Driver Guidelines*, [www.ni.com/idnet](http://www.ni.com/idnet), National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2004.

## Inputs and Outputs

To automate your lab, one of the first things you will have to tackle is data acquisition—the process of making measurements of physical phenomena and storing them in some coherent fashion. It's a vast technical field with thousands of practitioners, most of whom are hackers as we are. How do most of us learn about data acquisition? By doing it, plain and simple. Having a formal background in engineering or some kind of science is awfully helpful, but schools rarely teach the practical aspects of sensors and signals and so on. It's most important that you get the big picture—learn the pitfalls and common solutions to data acquisition problems—and then try to see where your situation fits into the grand scheme.

This chapter should be of some help because the information presented here is hard-won, practical advice, for the most part. The most unusual feature of this chapter is that it contains no LabVIEW programming information. There is much more to a LabVIEW system than LabVIEW programming! If you plan to write applications that support any kind of input/output (I/O) interface hardware, read on.

### Origins of Signals

Data acquisition deals with the elements shown in Figure 12.1. The physical phenomenon may be electrical, optical, mechanical, or something else that you need to measure. The sensor changes that phenomenon into a signal that is easier to transmit, record, and analyze—usually a voltage or current. Signal conditioning amplifies and filters the raw signal to prepare it for analog-to-digital conversion (ADC), which transforms the signal into a digital pattern suitable for use by your computer.

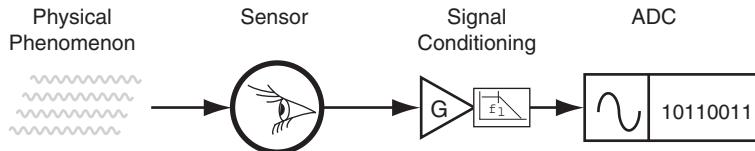


Figure 12.1 Elements of the data acquisition process.

### Transducers and sensors

A transducer converts one physical phenomenon to another; in our case, we're mostly interested in an electric signal as the output. For instance, a thermocouple produces a voltage that is related to temperature. An example of a transducer with a nonelectrical output is a liquid-in-glass thermometer. It converts temperature changes to visible changes in the volume of fluid. An elaboration on a transducer might be called a sensor. It starts with a transducer as the front end, but then adds signal conditioning (such as an amplifier), computations (such as linearization), and a means of transmitting the signal over some distance without degradation. Some industries call this a transmitter. Regardless of what you call it, the added signal conditioning is a great advantage in practical terms, because you don't have to worry so much about noise pickup when dealing with small signals. Of course, this added capability costs money and may add weight and bulk.

Figure 12.2 is a general model of all the world's sensors. If your instrument seems to have only the first couple of blocks, then it's probably a transducer. Table 12.1 contains examples of some sensors. The first one, example A, is a temperature transmitter that uses a thermocouple with some built-in signal conditioning. Many times you can handle thermocouples without a transmitter, but as we'll see later, you always need some form of signal conditioning. The second example, example B, is a pressure transmitter, a slightly more complex instrument. We came across the third example, example C, a magnetic field sensor,

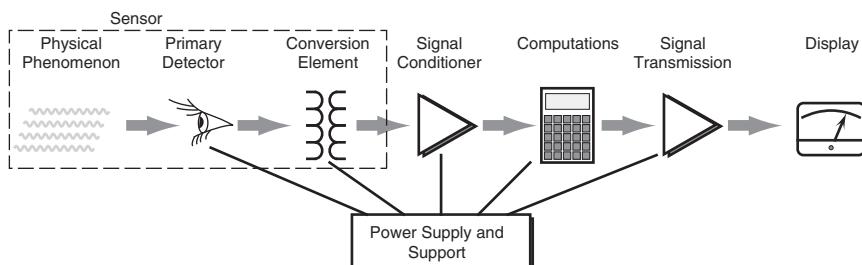


Figure 12.2 A completely general sensor model. Many times, your sensor is just a transducer that ends after the conversion element.

**TABLE 12.1 Three Practical Examples of Sensor Systems**

Block	Example A	Example B	Example C
Phenomenon	Temperature	Pressure	Magnetic field
Detector	—	Diaphragm displacement	Faraday rotation
Transducer	Thermocouple	LVDT	Laser and photodiode
Signal conditioner	Cold junction; amplifier	Demodulator	ADC
Computations	Linearize	Linearize; scale	Ratio; log; scale
Transmission	0–10 VDC	4–20 mA	RS-232 serial
Display	Analog meter	Analog meter	Computer system
Support	DC	DC (2-wire current loop)	DC; cooling

*Example A* is a temperature transmitter using a thermocouple, with cold-junction compensation, linearization, and analog output.

*Example B* is a pressure sensor using a linear variable differential transformer (LVDT) to detect diaphragm displacement, with analog output.

*Example C* is a magnetic field measurement using an optical technique with direct signal transmission to a computer. This represents a sophisticated state-of-the-art sensor system (3M Specialty Optical Fibers).

while doing some research. It uses an optical principle called Faraday rotation in which the polarization of light is affected by magnetic fields.

A detailed discussion of sensor technology covering all aspects of the physics of transducers and the practical matters of selecting the right sensor is beyond the scope of this book. For the purposes of data acquisition, there are several important things you need to know about each of your sensors:

- The nature of the signal it produces—voltage, amplitude range, frequency response, impedance, accuracy requirement, and so on—determines what kind of signal conditioning, analog-to-digital converter (ADC), or other hardware you might need.
- How susceptible is the sensor to noise pickup or loading effects from data acquisition hardware?
- How is the sensor calibrated with respect to the physical phenomenon? In particular, you need to know if it's nonlinear or if it has problems with repeatability, overload, or other aberrant behavior.
- What kind of power or other utilities it might require? This is often overlooked and sometimes becomes a show stopper for complex instruments!
- What happens if you turn off your data acquisition equipment while the sensor still has power applied? Will there be damage to any components?

When you start to set up your system, try to pick sensors and design the data acquisition system in tandem. They are highly interdependent.

When monitored by the wrong ADC, the world's greatest sensor is of little value. It is important that you understand the details of how your sensors work. Try them out under known conditions if you have doubts. If something doesn't seem right, investigate. When you call the manufacturer for help, it may well turn out that you know more about the equipment than the designers, at least in your particular application. In a later section, we'll look at a holistic approach to signals and systems.

Modern trends are toward smart sensors containing onboard microprocessors that compensate for many of the errors that plague transducers, such as nonlinearity and drift. Such instruments are well worth the extra cost because they tend to be more accurate and remove much of the burden of error correction from you, the user. For low-frequency measurements of pressure, temperature, flow, and level, the process control industry is rapidly moving in this direction. Companies such as Rosemount, Foxboro, and Honeywell make complete lines of smart sensors.

More complex instruments can also be considered sensors in a broader way. For instance, a digital oscilloscope is a sensor of voltages that vary over time. Your LabVIEW program can interpret this voltage waveform in many different ways, depending upon what the scope is connected to. Note that the interface to a sensor like this is probably GPIB or RS-232 serial communications rather than an analog voltage. That's the beauty of using a computer to acquire data: Once you get the hardware hooked up properly, all that is important is the signal itself and how you digitally process it.

## Actuators

An actuator, which is the opposite of a sensor, converts a signal (perhaps created by your LabVIEW program) into a physical phenomenon. Examples include electrically actuated valves, heating elements, power supplies, and motion control devices such as servomotors. Actuators are required any time you wish to control something such as temperature, pressure, or position. It turns out that we spend most of our time measuring things (the data acquisition phase) rather than controlling them, at least in the world of research. But control does come up from time to time, and you need to know how to use those analog and digital outputs so conveniently available on your interface boards.

Almost invariably, you will see actuators associated with feedback control loops. The reason is simple. Most actuators produce responses in the physical system that are more than just a little bit nonlinear and are sometimes unpredictable. For example, a valve with an electropneumatic actuator is often used to control fluid flow. The problem is that the flow varies in some nonlinear way with respect to the valve's position.

Also, most valves have varying degrees of nonrepeatability. They creak and groan and get stuck—*hysteresis* and *deadband* are formal terms for this behavior. These are real-world problems that simply can't be ignored. Putting feedback around such actuators helps the situation greatly. The principle is simple. Add a sensor that measures the quantity that you need to control. Compare this measurement with the desired value (the difference is called the *error*), and adjust the actuator in such a way as to minimize the error. Chapter 18, “Process Control Applications,” delves more deeply into this subject. The combination of LabVIEW and external loop controllers makes this whole situation easy to manage.

An important consideration for actuators is what sort of voltage or power they require. There are some industrial standards that are fairly easy to meet, such as 0 to 10 V dc or 4 to 20 mA, which are modest voltages and currents. But even these simple ranges can have added requirements, such as isolated grounds, where the signal ground is not the chassis of your computer. If you want to turn on a big heater, you may need large relays or contactors to handle the required current; the same is true for most high-voltage ac loads. Your computer doesn't have that kind of output, nor should it. Running lots of high power or high voltage into the back of your computer is not a pleasant thought! Try to think about these requirements ahead of time.

### Categories of signals

You measure a signal because it contains some type of useful information. Therefore, the first questions you should ask are, What information does the signal contain, and how is it conveyed? Generally, information is conveyed by a signal through one or more of the following signal parameters: state, rate, level, shape, or frequency content. These parameters determine what kind of I/O interface equipment and analysis techniques you will need.

Any signal can generally be classified as analog or digital. A digital, or binary, signal has only two possible discrete levels of interest—an active level and an inactive level. They're typically found in computer logic circuits and in switching devices. An analog signal, on the other hand, contains information in the continuous variation of the signal with respect to time. In general, you can categorize digital signals as either on/off signals, in which the state (on or off) is most important, or pulse train signals, which contain a time series of pulses. On/off signals are easily acquired with a digital input port, perhaps with some signal conditioning to match the signal level to that of the port. Pulse trains are often applied to digital counters to measure frequency, period, pulse width, or duty cycle. It's important to keep in mind that

digital signals are just special cases of analog signals, which leads to an important idea:

*Tip:* You can use analog techniques to measure and generate digital signals. This is useful when (1) you don't have any digital I/O hardware handy, (2) you need to accurately correlate digital signals and analog signals, or (3) you need to generate a continuous but changing pattern of bits.

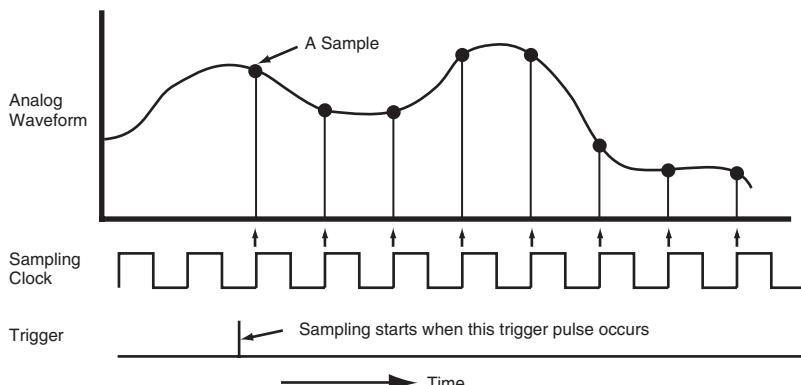
Among analog signal types are the dc signal and the ac signal. Analog dc signals are static or vary slowly with time. The most important characteristic of the dc signal is that information of interest is conveyed in the level, or amplitude, of the signal at a given instant. When measuring a dc signal, you need an instrument that can detect the level of the signal. The timing of the measurement is not difficult as long as the signal varies slowly. Therefore, the fundamental operation of the dc instrument is an ADC, which converts the analog electric signal into a digital number that the computer interprets. Common examples of dc signals include temperature, pressure, battery voltage, strain gauge outputs, flow rate, and level measurements. In each case, the instrument monitors the signal and returns a single value indicating the magnitude of the signal at a given time. Therefore, dc instruments often report the information through devices such as meters, gauges, strip charts, and numerical readouts.

*Tip:* When you analyze your system, map each sensor to an appropriate LabVIEW indicator type.

Analog **ac time domain** signals are distinguished by the fact that they convey useful information not only in the level of the signal, but also in how this level varies with time. When measuring a time domain signal, often referred to as a **waveform**, you are interested in some characteristics of the shape of the waveform, such as slope, locations and shapes of peaks, and so on. You may also be interested in its frequency content.

To measure the shape of a time domain signal with a digital computer, you must take a precisely timed sequence of individual amplitude measurements, or **samples**. These measurements must be taken closely enough together to adequately reproduce those characteristics of the waveform shape that you want to measure. Also, the series of measurements should start and stop at the proper times to guarantee that the useful part of the waveform is acquired. Therefore, the instrument used to measure time domain signals consists of an ADC, a sample clock, and a trigger. A sample clock accurately times the occurrence of each ADC.

Figure 12.3 illustrates the timing relationship among an analog waveform, a sampling clock, and a trigger pulse. To ensure that the desired portion of the waveform is acquired, you can use a trigger to start and/or stop the waveform measurement at the proper time

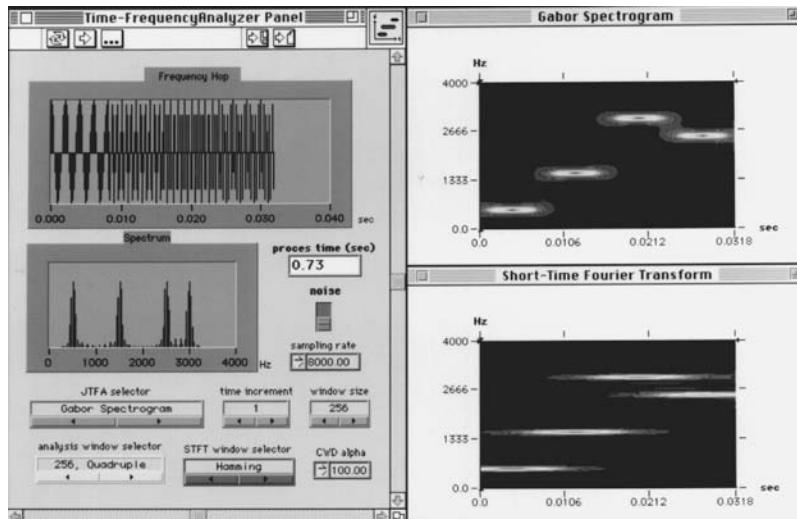


**Figure 12.3** An illustration of the relationship between an analog waveform and the sampling clock and trigger that synchronize an ADC.

according to some external condition. For instance, you may want to start acquisition when the signal voltage is moving in a positive direction through 0 V. Plug-in boards and oscilloscopes generally have trigger circuits that respond to such conditions.

Another way that you can look at an analog signal is to convert the waveform data to the **frequency domain**. Information extracted from frequency domain analysis is based on the frequency content of the signal, as opposed to the shape, or time-based characteristics of the waveform. Conversion from the time domain to the frequency domain on a digital computer is carried out through the use of a **Fast Fourier Transform (FFT)**, a standard function in the LabVIEW digital signal processing (DSP) function library. The **Inverse Fast Fourier Transform (IFFT)** converts frequency domain information back to the time domain. In the frequency domain, you can use DSP functions to observe the frequencies that make up a signal, the distribution of noise, and many other useful parameters that are otherwise not apparent in the time domain waveform. Digital signal processing can be performed by LabVIEW software routines or by special DSP hardware designed to do the analysis quickly and efficiently.

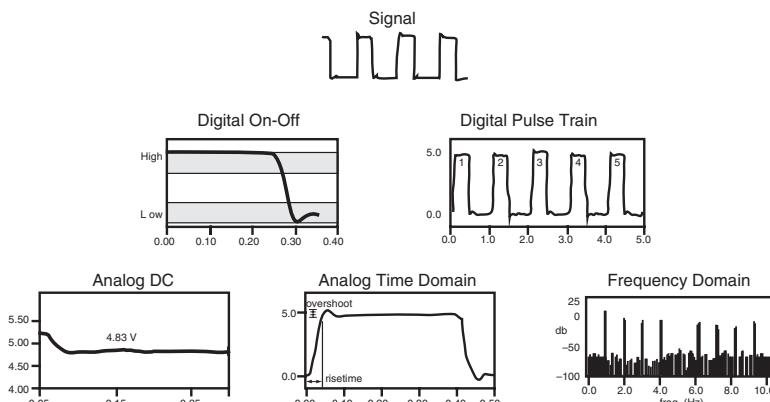
There is one more way to look at analog signals, and that is in the **joint time-frequency (JTF) domain** (Qian and Chen 1996). This is a combination of the two preceding techniques. JTF signals have an interesting frequency spectrum that varies with time. Examples are speech, sonar, and advanced modulation techniques for communication systems. The classic display technique for JTF analysis is the **spectrogram**, a plot of frequency versus time, of which LabVIEW has one of the very best, the Gabor spectrogram algorithm (Figure 12.4). You can order the LabVIEW Signal Processing Toolkit, which includes



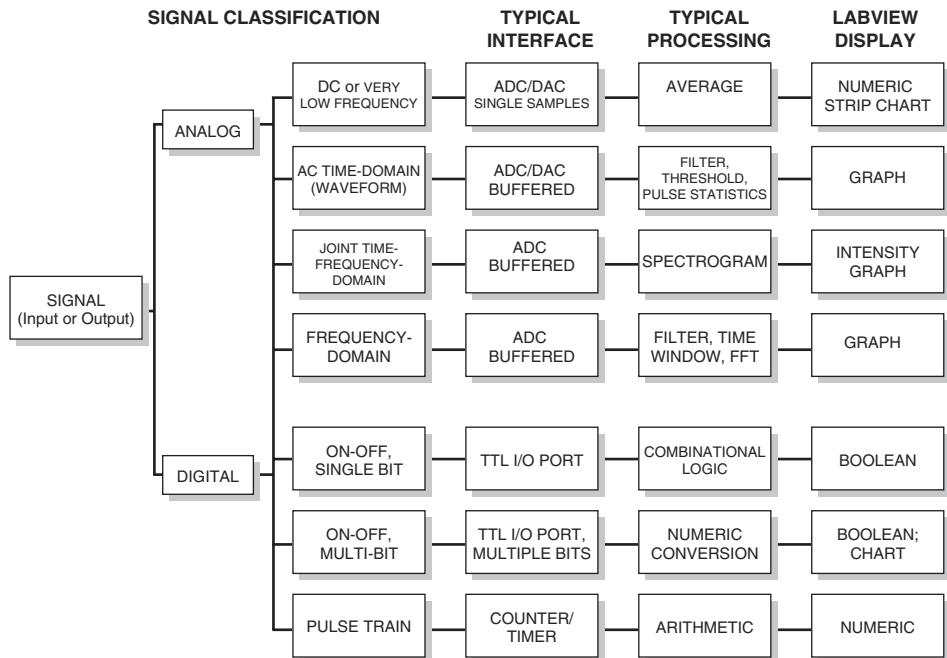
**Figure 12.4** These plots illustrate the striking differences between a joint time-frequency analysis plot of a chirp signal analyzed with the short-time FFT spectrogram and the same chirp signal analyzed with the Gabor spectrogram

a compiled application as well as the necessary VIs to do your own processing with the Gabor spectrogram, short-time FFT, and several others. It works with live data from a data acquisition board or any other source.

As Figure 12.5 shows, the signal classifications described in this section are not mutually exclusive. A single signal may convey more than



**Figure 12.5** Five views of a signal. A series of pulses can be classified in several different ways depending on the significance of the signal's time, amplitude, and frequency characteristics.



**Figure 12.6** This road map can help you organize your information about each signal to logically design your LabVIEW system.

one type of information. In fact, the digital on/off, pulse train, and dc signals are just simpler cases of the analog time domain signals that allow simpler measuring techniques.

The preceding example demonstrates how one signal can belong to many classes. The same signal can be measured with different types of instruments, ranging from a simple digital state detector to a complex frequency analysis instrument. This greatly affects how you choose signal conditioning equipment.

For most signals, you can follow the logical road map in Figure 12.6 to determine the signal's classification, typical interface hardware, processing requirements, and display techniques. As we'll see in coming sections, you need to characterize each signal to properly determine the kind of I/O hardware you'll need. Next, think about the signal attributes you need to measure or generate with the help of numerical methods. Then there's the user-interface issue—the part where LabVIEW controls and indicators come into play. Finally, you may have data storage requirements. Each of these items is directly affected by the signal characteristics.

## Connections

Professor John Frisbee is hard at work in his lab, trying to make a pressure measurement with his brand-new computer:

Let's see here.... This pressure transducer says it has a 0- to 10-V dc output, positive on the red wire, minus on the black. The manual for my data acquisition board says it can handle 0 to 10 V dc. That's no problem. Just hook the input up to channel 1 on terminals 5 and 6. Twist a couple of wires together, tighten the screws, run the data acquisition demonstration program, and voila! But what's this? My signal looks like ... junk! My voltmeter says the input is dc, 1.23 V, and LabVIEW seems to be working OK, but the display is really noisy. What's going on here?

Poor John. He obviously didn't read this chapter. If he had, he would have said, "Aha! I need to put a low-pass filter on this signal and then make sure that everything is properly grounded and shielded." What he needs is **signal conditioning**, and believe me, so do you. This world is chock full of noise sources and complex signals—all of them guaranteed to corrupt your data if you don't take care to separate the good from the bad.

There are several steps in designing the right signal conditioning approach for your application. First, remember that you need to know all about your sensors and what kind of signals they are supposed to produce. Second, you need to consider grounding and shielding. You may also need amplifiers and filters. Third, you can list your specifications and go shopping for the right data acquisition hardware.

### Grounding and shielding

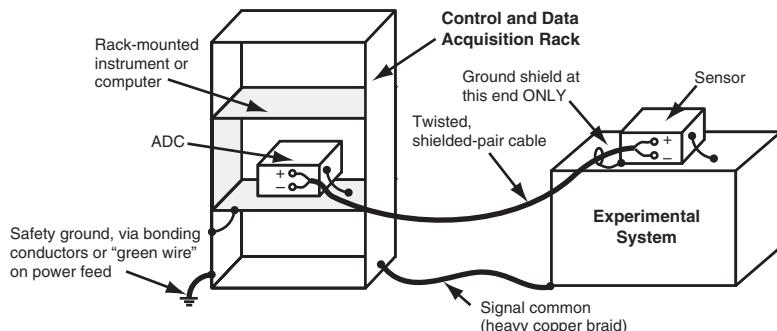
Noise sources are lurking everywhere, but by following some simple principles of grounding and shielding, you can eliminate most noise-related problems right at the source—the wiring. In fact, how you connect a sensor to its associated data acquisition hardware greatly affects the overall performance of the system (Gunn 1987; Morrison 1986; Ott 1988; White 1986). Another point on the subject: Measurements are inherently inaccurate. A good system design minimizes the distortion of the signal that takes place when you are trying to measure it—noise, nonlinear components, distortion, and so on. The wire between sensor and data acquisition can never improve the signal quality. You can only minimize the negatives with good technique. Without getting too involved in electromagnetic theory, we'll show you some of the recommended practices that instrumentation engineers everywhere use. Let's start with some definitions.

**Ground.** This is absolutely the most overused, misapplied, and misunderstood term in all electronics. First, there is the most hallowed

earth ground that is represented by the electrical potential of the soil underneath your feet. The green wire on every power cord, along with any metal framework or chassis with 120-V ac main power applied, is required by the National Electrical Code to be connected through a low-resistance path to the aforementioned dirt. There is exactly one reason for its existence: safety. Sadly, somehow we have come to believe that grounding our equipment will magically siphon away all sources of noise, as well as evil spirits. Baloney! Electricity flows only in closed circuits, or loops. You and your equipment sit upon earth ground as a bird sits upon a high-voltage wire. Does the bird know it's living at 34,000 V? Of course not; there is no complete circuit. This is not to say that connections to earth ground—which we will refer to as **safety ground** from here on—are unnecessary. You should always make sure that there is a reliable path from all your equipment to safety ground as required by code. This prevents accidental connections between power sources and metallic objects from becoming hazards. Such fault currents are shunted away by the safety ground system.

What we really need to know about is a reference potential referred to as **signal common**, or sometimes as a **return** path. Every time you see a signal or measure a voltage, always ask the question, "Voltage—with respect to what reference?" That reference is the signal common, which in most situations is not the same as safety ground. A good example is the negative side of the battery in your car. Everything electrical in the vehicle has a return connection to this common, which is also connected to the chassis. Note that there is no connection whatsoever to earth ground—the car has rubber tires that make fair insulators—and yet, the electrical system works just fine (except if it's very old or British).

In our labs, we run heavy copper braid, welding cable, or copper sheet between all the racks and experimental apparatus according to a grounding plan (Figure 12.7). A well-designed signal common can even



**Figure 12.7** Taking the system approach to grounding in a laboratory. Note the use of a signal common (in the form of heavy copper braid or cable) to tie everything together. All items are connected to this signal common.

be effective at higher frequencies at which second-order effects such as the **skin effect** and **self-inductance** become important. See Gunn (1987), Morrison (1986), Ott (1988), and White (1986) for instructions on designing a quality grounding system—it's worth its weight in aspirin.

**Electromagnetic fields.** Noise may be injected into your measurement system by electromagnetic fields, which are all around us. Without delving into Maxwell's equations, we present a few simple principles of electromagnetism that you use when connecting your data acquisition system.

**Principle 1.** Two conductors that are separated by an insulator form a **capacitor**. An electric field exists between the conductors. If the potential (voltage) of one conductor changes with time, a proportional change in potential will appear in the other. This is called *capacitive coupling* and is one way noise is coupled into a circuit. Moving things apart reduces capacitive coupling.

**Principle 2.** An electric field cannot enter a closed, conductive surface. That's why sitting in your car during a thunderstorm is better than sitting outside. The lightning's field cannot get inside. This kind of enclosure is also called a *Faraday cage*, or **electrostatic shield**, and is commonly implemented by a sheet of metal, screen, or braid surrounding a sensitive circuit. Electrostatic shields reduce capacitive coupling.

**Principle 3.** A time-varying magnetic field will induce an electric current only when a closed, conductive loop is present. Furthermore, the magnitude of the induced current is proportional to the intensity of the magnetic field and the area of the loop. This property is called *inductive coupling*, or **inductance**. Open the loop, and the current goes away.

**Principle 4.** Sadly, **magnetic shielding** is not so easy to design for most situations. This is so because the magnetic fields that we are most concerned about (low frequency, 50 /60 Hz) are very penetrating and require very thick shields of iron or even better magnetic materials, such as expensive mu-metal.

Here are some basic practices you need to follow to block the effects of these electromagnetic noise sources:

- Put sensitive, high-impedance circuitry and connections inside a metallic shield that is connected to the common-mode voltage (usually the low side, or common) of the signal source. This will block capacitive coupling to the circuit (principle 1), as well as the entry of any stray electric fields (principle 2).
- Avoid closed, conductive loops—intentional or unintentional—often known as **ground loops**. Such loops act as pickups for stray magnetic

fields (principle 3). If a high current is induced in, for instance, the shield on a piece of coaxial cable, then the resulting voltage drop along the shield will appear in series with the measured voltage.

- Avoid placing sensitive circuits near sources of intense magnetic fields, such as transformers, motors, and power supplies. This will reduce the likelihood of magnetic pickup that you would otherwise have trouble shielding against (principle 4).

Another unsuspected source of interference is your lovely color monitor on your PC. It is among the greatest sources of electrical interference known. Near the screen itself, there are intense electric fields due to the high voltages that accelerate the electron beam. Near the back, there are intense magnetic fields caused by the fly-back transformer which is used to generate high voltage. Even the interface cards in your computer, for instance, the video adapter and the computer's digital logic circuits, are sources of high-frequency noise. And never trust a fluorescent light fixture (or a politician).

**Radio-frequency interference (RFI)** is possible when there is a moderately intense RF source nearby. Common sources of RFI are transmitting devices such as walkie-talkies, cellular phones, commercial broadcast transmitters of all kinds, and RF induction heating equipment. Radio frequencies radiate for great distances through most nonmetallic structures, and really high (microwave) frequencies can even sneak in and out through cracks in metal enclosures. You might not think that a 75-MHz signal would be relevant to a 1-kHz bandwidth data acquisition system, but there is a phenomenon known as *parasitic detection* or *demodulation* that occurs in many places. Any time that an RF signal passes through a diode (a rectifying device), it turns into dc plus any low frequencies that may be riding on (modulating) the RF carrier. Likely parasitic detectors include all the solid-state devices in your system, plus any metal-oxide interfaces (such as dirty connections). As a licensed radio amateur, Gary knows that parasitic detection can occur in metal rain gutter joints, TV antenna connections, and stereo amplifier output stages. This results in neighbors screaming at him about how he cut up their TV or stereo.

When RFI strikes your data acquisition system, it results in unexplained noise of varying amplitude and frequency. Sometimes you can observe stray RF signals by connecting a wide-bandwidth oscilloscope to signal lines. If you see more than a few millivolts of high-frequency noise, suspect a problem. Some solutions to RFI are as follows:

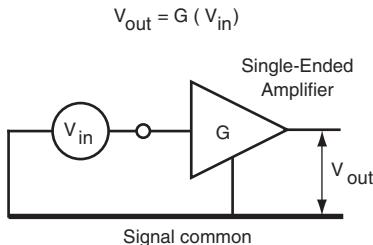
- Shield all the cables into and out of your equipment.
- Add RF rejection filters on all signal and power leads.
- Put equipment in well-shielded enclosures and racks.

- Keep known RF sources and cables far away from sensitive equipment.
- Keep the person with the walkie-talkie or cellular phone *out of your lab*.

**Other error sources.** **Thermojunction** voltages are generated any time two dissimilar metals come in contact with each other in the presence of a temperature gradient. This principle, known as the *Seebeck effect*, is the way in which a thermocouple generates its tiny signal. Problems occur in data acquisition when you attempt to measure dc signals that are in the microvolt to millivolt range, such as those from thermocouples and strain gauges. If you connect your instruments with wires, connectors, and terminal screws made of different metals or alloys, then you run the risk of adding uncontrolled thermojunction voltages to the signals—possibly to the tune of hundreds of microvolts. The most common case of thermojunction error that we have seen occurs when operators hook up thermocouples inside an experimental chamber and bring the wires out through a connector with pins made of copper or stainless steel. The thermocouple alloy wire meets the connector pin and forms a junction. Then the operators turn on a big heater inside the chamber, creating a huge temperature gradient across the connector. Soon afterward, they notice that their temperature readouts are way off. Those parasitic thermojunction voltages inside the connector are added to the data in an unpredictable fashion. Here are some ways to kill the thermojunction bugs:

- Make all connections with the same metallic alloy as the wires they connect.
- Keep all connections at the same temperature.
- Minimize the number of connections in all low-level signal situations.

**Differential and single-ended connections.** The means by which you connect a signal to your data acquisition system can make a difference in system performance, especially in respect to noise rejection. **Single-ended** connections are the simplest and most obvious way to connect a signal source to an amplifier or other measurement device (Figure 12.8). The significant hazard of this connection is that it is highly susceptible to noise pickup. Noise induced on any of the input wires, including the signal common, is added to the desired signal. Shielding the signal cable and being careful where you make connections to the signal common can improve the situation. Single-ended connections are most often used in wide-bandwidth systems such as oscilloscopes and video, RF, and fast pulse measurements in which low-impedance coaxial

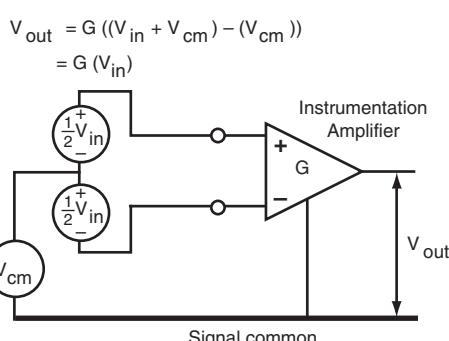


**Figure 12.8** A single-ended amplifier has no intrinsic noise rejection properties. You need to carefully shield signal cables and make sure that the signal common is noise-free as well.

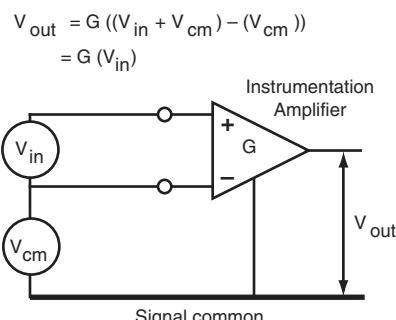
cables are the preferred means of transmission. These single-ended connections are also practical and trouble-free with higher-magnitude signals (say, 1 V or greater) that only need to travel short distances.

**Differential** connections depend on a pair of conductors where the voltage you want to measure, called the **normal-mode signal**, is the difference between the voltages on the individual conductors. Differential connections are used because noise pickup usually occurs equally on any two conductors that are closely spaced, such as a twisted pair of wires. So when you take the difference between the two voltages, the noise cancels but the difference signal remains. An **instrumentation amplifier** is optimized for use with differential signals, as shown in Figure 12.9. The output is equal to the gain of the amplifier times the difference between the inputs. If you add another voltage in series with both inputs, called the **common-mode signal**, it cancels just like the noise pickup. The optimum signal source is a **balanced** signal where

#### Proper differential amplifier application



#### More common application (Poorer common-mode rejection)



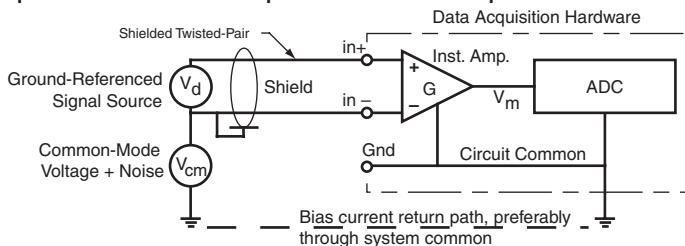
**Figure 12.9** Instrumentation amplifiers effectively reject common-mode ( $V_{\text{cm}}$ ) signals, such as noise pickup, through cancellation. The normal-mode (differential) signal is amplified.

the signal voltage swings symmetrically with respect to the common-mode voltage. Wheatstone bridge circuits and transformers are the most common balanced sources.

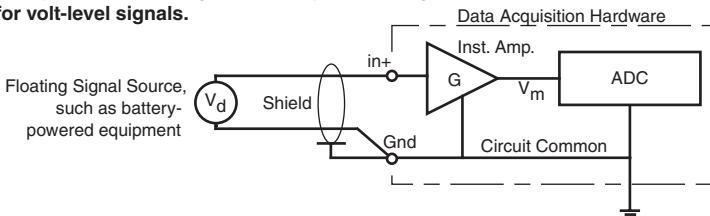
The common-mode voltage can't be infinitely large, though, since every amplifier has some kind of maximum input voltage limit, which is usually less than 10 V—watch out for overloads! Also, beware that the common-mode rejection ratio (CMRR) of an amplifier generally decreases with frequency. Don't count on an ordinary amplifier to reject very intense RF signals, for instance. Differential inputs are available on most low-frequency instrumentation such as voltmeters, chart recorders, and plug-in boards, for example, those made by National Instruments. *Rule: Always use differential connections, except when you can't.*

In Figure 12.10, you can see some typical signal connection schemes. Shielded cable is always recommended to reduce capacitive coupling

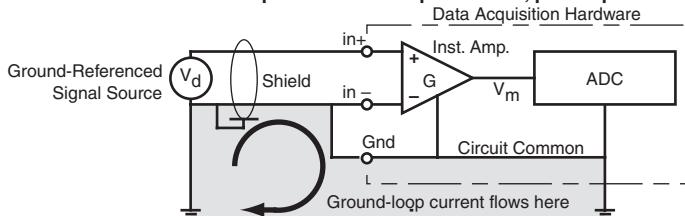
**Acceptable use of differential inputs. Strive for this setup.**



**Acceptable use of single-ended inputs. The signal cable could also be coax for volt-level signals.**



**Typical misuse of differential inputs. Ground-loop is formed, picks up noise.**



**Figure 12.10** Proper use of differential and single-ended signal connections is shown in the top two parts of the figure. The bottom part is that all-too-common case in which a ground loop is formed, enhancing the pickup of noise borne by magnetic fields.

and electric field pickup. Twisted, shielded pairs are best, but coaxial cable will do in situations where you are careful not to create the dreaded ground loop that appears in the bottom segment of the figure.

### Why use amplifiers or other signal conditioning?

As you can see, the way you hook up your sensors can affect the overall performance of your data acquisition system. But even if the grounding and shielding are properly done, you should consider **signal conditioning**, which includes **amplifiers** and **filters**, among other things, to reduce the noise relative to the signal level.

Amplifiers improve the quality of the input signal in several ways. First, they boost the amplitude of smaller signals, improving the resolution of the measurements. Second, they offer increased driving power (lower output impedance), which keeps such things as ADCs from loading the sensor. Third, they provide differential inputs, a technique known to help reject noise. An improvement on differential inputs, an **isolation amplifier**, requires no signal common at the input whatsoever.\* Isolation amplifiers are available with common-mode voltage ranges up to thousands of volts.

Amplifiers are usually essential for microvolt signals such as those from thermocouples and strain gauges. In general, try to amplify your low-level signal as close to the physical phenomenon itself as possible. Doing this will help you increase the **signal-to-noise ratio (SNR)**. The signal-to-noise ratio is defined as

$$\text{SNR} = 20 \log \left( \frac{V_{\text{sig}}}{V_{\text{noise}}} \right)$$

where  $V_{\text{sig}}$  is the signal amplitude and  $V_{\text{noise}}$  is the noise amplitude, both measured in volts rms. The  $20 \log()$  operation converts the simple ratio to **decibels (dB)**, a ratiometric system used in electrical engineering, signal processing, and other fields. Decibels are convenient units for gain and loss computations. For instance, an SNR or gain of 20 dB is the same as a ratio of 10 to 1; 40 dB is 100 to 1; and so forth. Note that the zero noise condition results in an infinite SNR. May you one day achieve this.

---

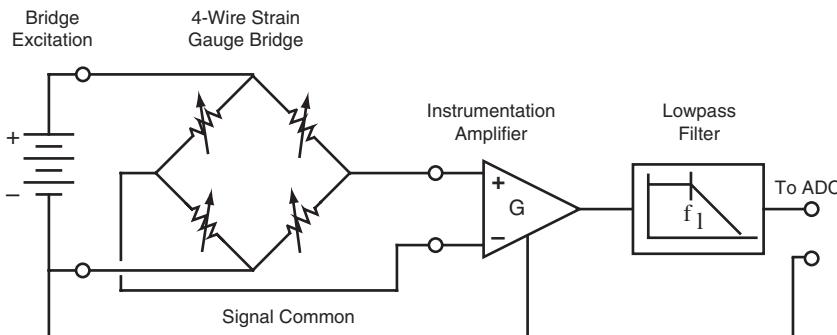
\*Manufacturers say you don't need a ground reference, but experience in the field says it ain't so. George Wells of JPL reports that Analog Devices 5B modules sometimes couple large common-mode voltages to adjacent modules. This may be so because of internal transformer orientation. Other brands may have their own problems. Your best bet is to provide a proper ground to all signals.

Amplifiers are available in stand-alone modular form, such as those from Action Instruments and Moore Products that mount on DIN rails, or in chassis-mounted modules, available from Preston and Ectron. Another choice is a module such as the National Instruments SCXI-1120, which includes an isolation amplifier and a low-pass filter on each of eight channels, followed by a multiplexer that routes the selected channel to a common ADC input. Most plug-in data acquisition boards include a programmable-gain instrumentation amplifier (PGIA). You change the gain by sending a command to the board, and the gain can be changed at any time—even while you are rapidly scanning through a set of channels. The difference between an onboard amplifier and an external amplifier is that the onboard amplifier must respond quickly to large changes in signal level (and perhaps gain setting) between channels. Errors can arise due to the **settling time**, which is the time it takes for the amplifier output to stabilize within a prescribed error tolerance of the final value. If you scan a set of channels at top speed with the amplifier at high gain, there is the possibility of lost accuracy. National Instruments has gone to great pains to ensure that its custom-designed PGAs settle very rapidly, and it guarantees full accuracy at all gains and scanning speeds. But if you can afford it, the amplifier-per-channel approach remains the best choice for high-accuracy data acquisition.

Filters are needed to reject undesired signals, such as high-frequency noise, and to provide **antialiasing** (discussed later in this chapter). Most of the time, you need low-pass filters which reject high frequencies while passing low frequencies including dc. The simplest filters are made from resistors and capacitors (and sometimes inductors). **Active filters** combine resistors and capacitors with operational amplifiers to enhance performance. **Switched capacitor filters** are a modern alternative to these ordinary analog filters. They use arrays of small capacitors that are switched rapidly in and out of the circuit, simulating large resistors on a much smaller (integrated circuit) scale.

A wide variety of external filters are available in modular form for mounting on printed-circuit boards and DIN rails. They are also available in plug-in modules, such as the National Instruments SCXI-1141 nine-channel elliptic low-pass filter module, or in rack-mounted enclosures, such as those made by Frequency Devices, Precision Filters, and TTE. Alternatively, you can use the aforementioned SCXI-1120 with an amplifier and filter for each channel. (You might have guessed by now that we like the SCXI-1120 module. We use it a lot because it has excellent common-mode rejection, filtering, and accuracy.)

Using a signal conditioning system outside of your computer also enhances safety. If a large overload should occur, the signal conditioner will take the hit rather than pass it directly to the backplane of your



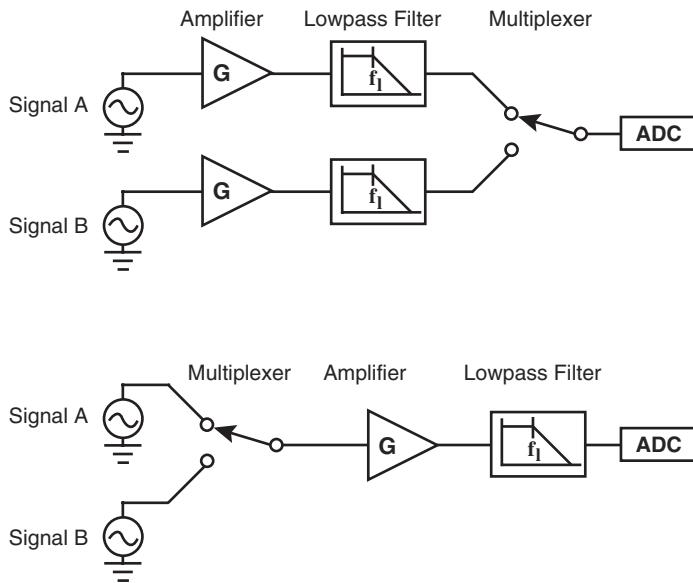
**Figure 12.11** Schematic of a signal conditioner for a strain gauge, a bridge-based transducer that requires excitation. This signal conditioner has an instrumentation amplifier with high gain, followed by a low-pass filter to reject noise.

computer (and maybe all the way to the mouse or keyboard!). A robust amplifier package can easily be protected against severe overloads through the use of transient absorption components (such as varistors, zener diodes, and spark gaps), limiting resistors, and fuses. Medical systems are covered by federal and international regulations regarding isolation from stray currents. *Never connect electronic instruments to a live subject (human or otherwise) without a properly certified isolation amplifier and correct grounding.*

Special transducers may require **excitation**. Examples are resistance temperature detectors (RTDs), thermistors, potentiometers, and strain gauges, as depicted in Figure 12.11. All these devices produce an output that is proportional to an excitation voltage or current as well as the physical phenomenon they are intended to measure. Thus, the excitation source can be a source of noise and drift and must be carefully designed. Modular signal conditioners, such as the Analog Devices 5B series, and SCXI hardware from National Instruments include high-quality voltage or current references for this purpose.

Some signal conditioning equipment may also have built-in **multiplexing**, which is an array of switching elements (relays or solid-state analog switches) that route many input signals to one common output. For instance, using SCXI equipment, you can have hundreds of analog inputs connected to one multiplexer at a location near the experiment. Then only one cable needs to be run back to a plug-in board in your computer. This drastically reduces the number and length of cables. Most plug-in boards include multiplexers.

Multiplexing can cause some interesting problems when not properly applied, as we mentioned in regard to the amplifier settling time. Compare the two signal configurations in Figure 12.12. In the top configuration, there is an amplifier and filter per channel, followed by the



**Figure 12.12** The preferred amplifier and filter per channel configuration (top); an undesirable, but cheaper, topology (bottom). Watch your step if your system looks like the latter.

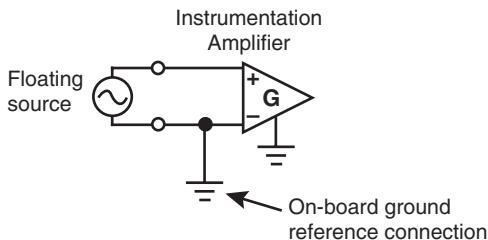
multiplexer which ultimately feeds the ADC. The SCXI-1120, SCXI-1102, and many packaged data acquisition systems are designed in this way. It is the preferred arrangement because each filter output faithfully follows its respective input signal without disturbances from switching transients. But in the bottom configuration, the amplifier and filter are located after the multiplexer. This saves money but is undesirable because each time the multiplexer changes channels, the very sluggish low-pass filter must settle to the voltage present at the new channel. As a result, you must scan the channels at a very slow rate unless you disable the low-pass filter, thus giving up its otherwise useful properties. The SCXI-1100 32-channel multiplexer module is designed in this way and has caught some users unaware.

**Practical tips on connecting input signals.** Let's look at a few common input signal connection situations that you're likely to come across. We recommend that you refer to the manual for your signal conditioning or plug-in board before making any connections. There may be particular options or requirements that differ from those in this tutorial.

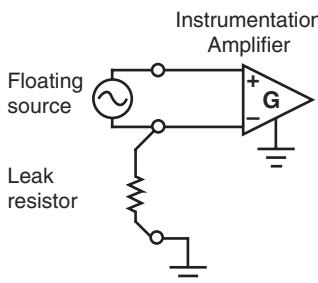
**Ground-referenced analog inputs.** If one side of your signal has a direct connection to a reliable ground, use a differential input to avoid ground loops and provide common-mode noise rejection.

**Floating analog inputs.** Battery-powered equipment and instruments with isolated outputs do not supply a return connection to signal ground. This is generally good, since a ground loop is easy to avoid. You can connect such a signal to a *referenced single-ended (RSE) input* (Figure 12.13A), an input configuration available on most data acquisition boards, and called RSE in the National Instruments manuals. Common-mode noise rejection will be acceptable in this case. You can also use differential inputs, but note that they require the addition of a pair of *leak resistors* (Figure 12.13B and C). Every amplifier injects

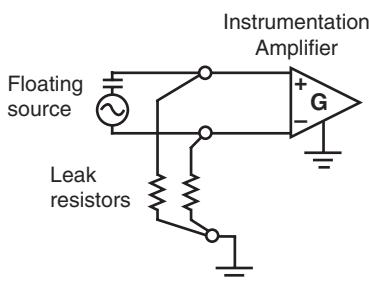
A. Referenced single-ended connection



B. Differential connection – DC-coupled source



C. Differential connection – AC-coupled source



**Figure 12.13** Floating source connections: (A) This is the simplest and most economical; (B) this is similar but may have noise rejection advantages; and (C) this is required for ac-coupled sources.

a small *bias current* from its inputs back into the signal connections. If there is no path to ground, as is the case for a true floating source, then the input voltage at one or both inputs will float to the amplifier's power supply rail voltage. The result is erratic operation because the amplifier is frequently saturated—operating out of its linear range. This is a sneaky problem! Sometimes the source is floating and you don't even know it. Perhaps the system will function normally for a few minutes after power is turned on, and then it will misbehave later. Or, touching the leads together or touching them with your fingers may discharge the circuit, leading you to believe that there is an intermittent connection. Yikes!

*Rule: Use an ohmmeter to verify the presence of a resistive path to ground on all signal sources.*

What is the proper value for a leak resistor? The manual for your input device may have a recommendation. In general, if the value is too low, you lose the advantage of a differential input because the input is tightly coupled to ground through the resistor. You may also overload the source in the case where leak resistors are connected to both inputs. If the value is too high, additional dc error voltages may arise due to *input offset current drift* (the input bias currents on the two inputs differ and may vary with temperature). A safe value is generally in the range of 1 to 100 k $\Omega$ . If the source is truly floating, such as battery-powered equipment, then go ahead and directly ground one side.

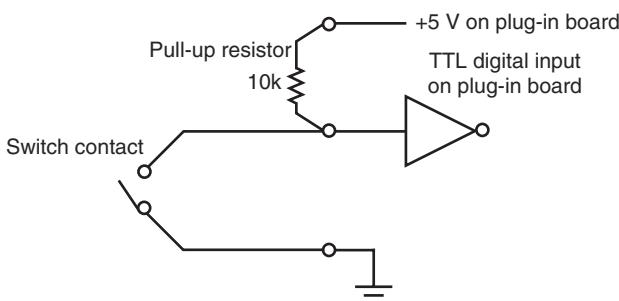
**Thermocouples.** Thermocouples have very low source impedance and low signal voltage, and they may have either floating or grounded junctions. For an excellent tutorial on thermocouples, refer to Omega Engineering's *Temperature Handbook* (2001). For general information on connecting thermocouples to computer I/O, consult National Instruments' Web site. We generally prefer floating junctions, in which the welded joint between the thermocouple wires is isolated from the surrounding sheath or other nearby metallic items. This removes another chance of a ground loop and results in a floating dc-coupled source as previously described. Always use differential connections to reject noise on this and other low-level signals, and be sure to use a leak resistor—perhaps 1 k $\Omega$  or so—on one of the inputs. You must also take care to use extension wire and connectors of the same alloy as that of the thermocouple element. Finally, the connections to the signal conditioning must be *isothermal*; that is, all connections are at the same temperature.

Thermocouples also require a *reference junction* measurement to compensate for the voltage produced at the junction between the thermocouple wires and copper connections. This may be provided by an additional thermocouple junction immersed in a reference temperature bath or, more commonly, by measuring the temperature at the reference temperature block, where the thermocouple alloy transitions to copper,

typically near the amplifier. All modern signal conditioners include an isothermal reference temperature feature. The LabVIEW DAQ library includes VIs for making the corrections and performing linearization.

**AC signals.** If you want to measure the RMS value of an ac waveform, you have several choices: Use an external AC to DC converter, digitize the waveform at high speed and do some signal processing in LabVIEW, use a plug-in digital multimeter board such as the National Instruments 4030, or use a benchtop DMM with a GPIB connection. For low-frequency voltage measurements, your best bet is a true RMS (TRMS) converter module, such as those made by Action Instruments and Ohio Semitronics (which also make single- and three-phase ac power transmitters). The input can be any ac waveform up to a few kilohertz and with amplitudes as high as 240 V RMS. For higher frequencies, a more sophisticated converter is required, and you may well have to build it yourself, using an IC such as the Linear Technology LT1088 RMS to DC converter, which operates from dc to 100 MHz. To measure low-frequency current, use a current transformer and an ac converter module as before. If the frequency is very high, special current transformers made by Pearson Electronics are available and can feed a wideband RMS to DC converter. All these transformers and ac converters give you signals that are isolated from the source, making connections to your DAQ board simple. When the shape of the waveform is of interest, you generally connect the signal directly to the input of a sufficiently fast ADC or digital oscilloscope, and then you acquire large buffers of data for analysis.

**Digital inputs.** Digital on/off signals may require conditioning, depending upon their source. If the source is an electronic digital device, it probably has TTL-compatible outputs and can be connected directly to the digital I/O port of a plug-in board. Check the manuals for the devices at both ends to ensure that both the levels and current requirements match. Detecting a contact closure (a switch) requires a *pull-up resistor* at the input (Figure 12.14) to provide the high level when the switch is

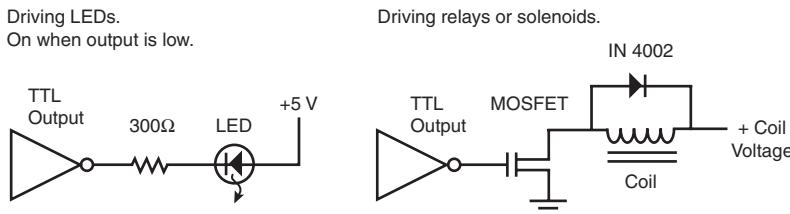


**Figure 12.14** Contact closures are sensed by a TTL input with the help of an external pull-up resistor.

open. If your source produces a voltage outside of the 0- to 5-VDC range of a typical plug-in board, additional signal conditioning is required. You can buy a board such as one of the National Instruments SSR series which is compatible with high ac and dc voltages. Even better, the SCXI-1162HV gives you 32 channels of isolated high-voltage inputs.

**Outputs need signal conditioning, too.** To drive actuators of one type or another, you may need signal conditioning. As with input signals, isolation is one of the major considerations, along with requirements for extra drive capability that many simple output devices can't handle directly.

For digital outputs, most plug-in boards offer simple TTL logic drivers, which swing from 0 to about +3 V, depending on the load (you need to consult the specifications for your hardware to make sure). If you want to drive a solenoid valve, for instance, much more current is needed—perhaps several amperes at 24 V dc or even 120 V ac. In such cases, you need a relay of some type. Electromechanical relays are simple and inexpensive and have good contact ratings, but sometimes they require more coil current than a TTL output can supply. Sensitive relays, such as reed relays, are often acceptable. Solid-state relays use silicon controlled rectifiers (SCRs) or triacs to control heavier loads with minimal control current requirements. Their main limitation is that most units are only usable for ac circuits. Many types of modular I/O systems have options for these higher voltages and currents, such as the SCXI-1161 with eight power relays, which you can connect directly to your plug-in board. If you need to drive other types of logic or require large output voltages and/or currents at high speed, then a special interface circuit may have to be custom-designed. Figure 12.15 shows a couple of simple options for driving light-emitting diodes (LEDs) and heavier dc loads, such as solenoids, using MOSFETs to handle the power. National Instruments sells a variety of digital signal conditioners that plug right into its boards; use them first because they're convenient.



**Figure 12.15** Simple circuits you can use with digital outputs on MIO and DIO series boards. Drive low-current loads like LEDs directly (left). MOSFETs are available in a wide range of current and voltage ratings for driving heavy dc loads (right).

One warning is in order regarding digital outputs. When you turn on a digital output device, be it a plug-in board or an external module, which output state will it produce—on, off, or high-impedance (disconnected)? Can it produce a brief but rapid pulse train? We've seen everything, and you will, too, given time. Connect an oscilloscope to each output and see what happens when you turn the system on and off. If the result is unsatisfactory (or unsafe!), you may need a switch or relay to temporarily connect the load to the proper source. This warning applies to *analog* outputs, too.

Most of the analog world gets by with control voltages in the range of  $\pm 10$  V or less and control currents of 20 mA or less. These outputs are commonly available from most I/O boards and modular signal conditioners. If you need more current and/or voltage, you can use a power amplifier. Audio amplifiers are a quick and easy solution to some of these problems, although most have a high-pass filter that rolls off the dc response (that is, they can produce no dc output). Many commercial dc power supplies have analog inputs and are good for low-frequency applications. To control big ac heaters, motors, and other heavy loads, look to triac-based power controllers. They permit an ordinary analog voltage or current input to control tens or even hundreds of kilowatts with reasonable cost and efficiency. One thing to watch out for is the need for isolation when you get near high-power equipment. An isolation amplifier could save your equipment if a fault occurs; it may also eliminate grounding problems which are so prevalent around high-power systems.

### Choosing the right I/O subsystem

As soon as your project gets underway, make a list of the sensors you need to monitor along with any actuators you need to drive. Having exact specifications on sensors and actuators will make your job much easier. As an instrumentation engineer, Gary spends a large part of his time on any project just collecting data sheets, calling manufacturers, and peering into little black boxes, trying to elucidate the important details needed for interfacing. Understanding the application is important, too. The fact that a pressure transducer will respond in 1  $\mu$ s doesn't mean that your system will actually have microsecond dynamic conditions to measure. On the other hand, that superfast transducer may surprise you by putting out all kinds of fast pulses because there are some little bubbles whizzing by in the soup. Using the wrong signal conditioner in either case can result in a phenomenon known as *bogus data*.

Since you are a computer expert, fire up your favorite spreadsheet or database application and make up an instrument list. The process

control industry goes so far as to standardize on a format known as *instrument data sheets*, which are used to specify, procure, install, and finally document all aspects of each sensor and actuator. Others working on your project, particularly the quality assurance staff, will be very happy to see this kind of documentation. For the purposes of designing an I/O subsystem, your database might include these items:

- Instrument name or identifier
- Location, purpose, and references to other drawings such as wiring and installation details
- Calibration information: engineering units (such as psig) and full-scale range
- Accuracy, resolution, linearity, and noise, if significant
- Signal current, voltage, or frequency range
- Signal bandwidth
- Isolation requirements
- Excitation or power requirements: current and voltage

To choose your I/O subsystem, begin by sorting the instrument list according to the types of signal and other basic requirements. Remember to add plenty of spare channels! Consider the relative importance of each instrument. If you have 99 thermocouples and 1 pressure gauge, your I/O design choice will certainly lean toward accommodating thermocouples. But that pressure signal may be the single most important measurement in the whole system; don't try to adapt its 0- to 10-V output to work with an input channel that is optimized for microvolt thermocouple signals. For each signal, determine the minimum specifications for its associated signal conditioner. Important specifications are as follows:

- Adjustability of zero offset and gain
- Bandwidth—minimum and maximum frequencies to pass
- Filtering—usually antialiasing low-pass filters
- Settling time and phase shift characteristics
- Accuracy
- Gain and offset drift with time and temperature (very important)
- Excitation—built in or external?
- For thermocouples, cold-junction compensation
- Linearization—may be better performed in software

Next, consider the physical requirements. Exposure to weather, high temperatures, moisture and other contamination, or intense electromagnetic interference may damage unprotected equipment. Will the equipment have to work in a hostile environment? If so, it must be in a suitable enclosure. If the channel count is very high, having many channels per module could save both space and money. Convenience should not be overlooked: Have you ever worked on a tiny module with itty-bitty terminal screws that are deeply recessed into an overstuffed terminal box? This is a practical matter; if you have lots of signals to hook up, talk this over with the people who will do the installation.

If your company already has many installations of a certain type or I/O, that may be an overriding factor, as long as the specifications are met. The bottom line is always cost. Using excess or borrowed equipment should always be considered when money is tight. You can do a cost-per-channel analysis, if that makes sense. For instance, using a multifunction board in your computer with just a few channels hooked up through the Analog Devices 5B series signal conditioners is very cost-effective. But if you need 100 channels, using SCXI would certainly save money over using the Analog Devices 5Bs. (See Figure 12.16.)

**Remote and distributed I/O.** Your sensors may not be located close to your computer system, or they may be in an inaccessible area—a hazardous enclosure or an area associated with a high-voltage source.

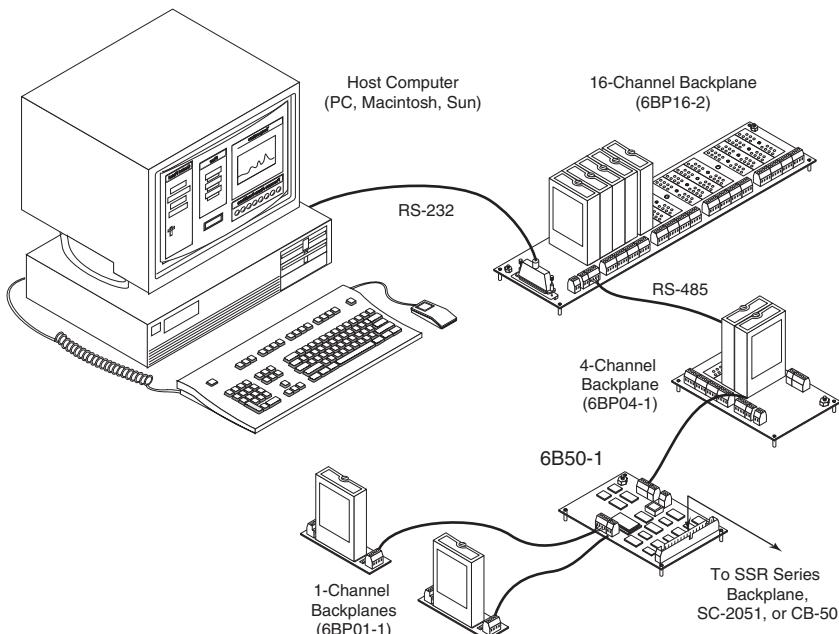


**Figure 12.16** EG&G Automotive Research (San Antonio, Texas) has the type of industrial environment that makes effective use of SCXI signal conditioning products. (*Photo courtesy of National Instruments and EG&G Automotive Research.*)

In such cases, **remote I/O** hardware is appropriate. It makes sense to locate the acquisition hardware close to groups of sensors—*remote* from the computer—because it saves much signal wiring which in turn reduces cost and reduces the chance of noise pickup. Many remote I/O systems are commercially available with a variety of communication interfaces. If more than one I/O subsystem can be connected to a common communication line, it is often referred to as **distributed I/O**.

National Instruments carries the **Analog Devices 6B series** distributed I/O system. This family includes a wide variety of analog and digital I/O modules that plug into backplanes that hold 1, 4, or 16 modules. Backplanes are connected to the host computer via RS-232 or RS-485 serial lines, as shown in Figure 12.17. Most computers have RS-232 built in, and it is adequate for short distances. For RS-485, you can install an AT-485 RS-485 interface and gain the ability to extend your reach to 4000 ft. LabVIEW drivers are available for all platforms. See the National Instruments catalog for details.

A similar distributed I/O system is made by **Opto-22**. As with the Analog Devices 6B series, you choose from a variety of analog and digital I/O modules that are installed in backplanes of various types and sizes. A *brain board* (model B1 for digital and B2 for analog I/O)



**Figure 12.17** The Analog Devices 6B series, available through National Instruments, provides distributed I/O capability with serial communications.

connects one or more backplanes to an RS-422 serial line running the Optomux protocol. A free LabVIEW driver is available from National Instruments. We've found Optomux equipment to be a simple, reliable, low-cost solution for connecting remote sensors and systems around the lab.

SCXI equipment can be remotely operated over serial links, specifically RS-232 or RS-485, up to 1.2 km. You can use the SCXI-2000 four-slot chassis with built-in communications, or the SCXI-1124 communications module in any of the standard SCXI chasses, and plug in all the usual modules. SCXI, of course, maintains transparency with regard to the NI DAQ driver configurations, regardless of the fact that it's connected over a serial link. You may sacrifice some performance, however.

National Instruments has a new line of modular I/O called **FieldPoint**. It's designed primarily for industrial process control applications, as is the Analog Devices 6B module. Dozens of I/O modules are available, covering every type of analog and digital signal. It's ideal when you need to put the I/O interface close to the process, concentrating those interfaces in clusters that are linked over a network. FieldPoint features several networking options: RS-232 (including wireless), RS-485, TCP/IP over Ethernet (10 or 100 Mb/s), and Foundation Fieldbus. There's also a smart FieldPoint controller module that works as a LabVIEW RT target over Ethernet. Thus you can create your own real-time industrial controller—programmed in LabVIEW rather than ladder logic or C—that snaps on a DIN rail. Now that's versatile. The only limitation with FieldPoint at this time is that the driver software is available only for Windows.

### Network everything!

Ethernet connections are just about everywhere these days, including wireless links. There are many ways that you can take advantage of these connections when configuring a distributed measurement and control system, courtesy of National Instruments. Here are a few configurations and technologies to consider.

- Data can be published via **DataSockets**, a cross-platform technology supported by LabVIEW that lets you send any kind of data to another LabVIEW-based system or to a Web browser. It's as easy as writing data to a file, except it is being spun out on the Web. And it's as easy to access as a file, too.
- **Remote data acquisition (RDA)** is an extension of the NI DAQ driver for Windows where a DAQ board plugged into one computer is accessible in real time over the network on another computer.

For instance, you could have a dedicated laboratory system connected to an experiment, while back in your office you can configure and run data acquisition operations (from LabVIEW, of course) as if the DAQ board were plugged into your desktop machine.

- Ethernet adapters, such as the GPIB-ENET, are available for GPIB instruments. Also, many new rack-and-stack instruments are featuring Ethernet interfaces. That lets you extend the GPIB bus as far as you want—perhaps around the world—with nothing more than an Ethernet jack required on your computer.

Now you have your signals connected to your computer. In Chapter 13, we'll take a closer look at how those signals are sampled or digitized for use in LabVIEW.

## Bibliography

- Beckwith, Thomas G. and R. D. Marangoni, *Mechanical Measurements*, Addison-Wesley, Reading, Massachusetts, 1990. (ISBN 0-201-17866-4) Gunn, Ronald, "Designing System Grounds and Signal Returns," *Control Engineering*, May, 1987.
- Lancaster, Donald, *Active Filter Cookbook*, Howard W. Sams & Co., Indianapolis, 1975. (ISBN 0-672-21168-8) Lipshitz, Stanley P., R. A. Wannamaker, and J. Vanderkooy, "Quantization and Dither: A Theoretical Survey," *J. Audio Eng. Soc.*, 40(5):355-375 (1992).
- Morrison, Ralph, *Grounding and Shielding Techniques in Instrumentation*, Wiley-Interscience, New York, 1986.
- Norton, Harry R. *Electronic Analysis Instruments*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992. (ISBN 0-13-249426-4) Omega Engineering, Inc., *Temperature Handbook*, Stamford, Connecticut, 2000. (Available free by calling 203-359-1660 or 800-222-2665.)
- Ott, Henry W., *Noise Reduction Techniques in Electronic Systems*, John Wiley & Sons, New York, 1988. (ISBN 0-471-85068-3) Pallas-Areny, Ramon and J. G. Webster, *Sensors and Signal Conditioning*, John Wiley & Sons, New York, 1991. (ISBN 0-471-54565-1) Qian, Shie and Dapang Chen, *Joint Time-Frequency Analysis—Methods and Applications*.
- Prentice-Hall, Englewood Cliffs, New Jersey, 1996. (ISBN 0-13-254384-2. Call Prentice-Hall at 800-947-7700 or 201-767-4990.)
- Sheingold, Daniel H., *Analog-Digital Conversion Handbook*, Prentice-Hall, Englewood Cliffs, New Jersey, 1986. (ISBN 0-13-032848-0) Steer, Robert W., Jr., "Anti-aliasing Filters Reduce Errors in ADC Converters," *EDN*, March 30, 1989.
- 3M Specialty Optical Fibers, *Fiber Optic Current Sensor Module* (product information and application note), West Haven, Connecticut, (203) 934-7961.
- White, Donald R. J. *Shielding Design Methodology and Procedures*, Interference Control Technologies, Gainesville, Virginia, 1986. (ISBN 0-932263-26-7)

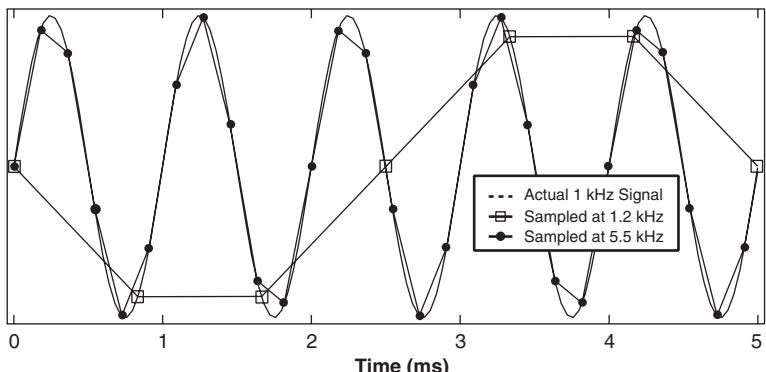
# Sampling Signals

Up until now, we've been discussing the real (mostly analog) world of signals. Now it's time to digitize those signals for use in LabVIEW. By definition, **analog** signals are **continuous-time**, **continuous-value** functions. That means they can take on any possible value and are defined over all possible time resolutions. (By the way, don't think that digital pulses are special; they're just analog signals that happen to be square waves. If you look closely, they have all kinds of ringing, noise, and slew rate limits—all the characteristics of analog signals.)

An **analog-to-digital converter** (ADC) samples your analog signals on a regular basis and converts the amplitude at each sample time to a digital value with finite resolution. These are termed **discrete-time**, **discrete-value** functions. Unlike their analog counterparts, discrete functions are defined only at times specified by the sample interval and may only have values determined by the resolution of the ADC. In other words, when you digitize an analog signal, *you have to approximate*. How *much* you can throw out depends on your signal and your specifications for data analysis. Is 1 percent resolution acceptable? Or is 0.0001 percent required? And how fine does the temporal resolution need to be? One second? Or 1 ns? Please be realistic. Additional amplitude and temporal resolution can be *expensive*. To answer these questions, we need to look at this business of sampling more closely.

## Sampling Theorem

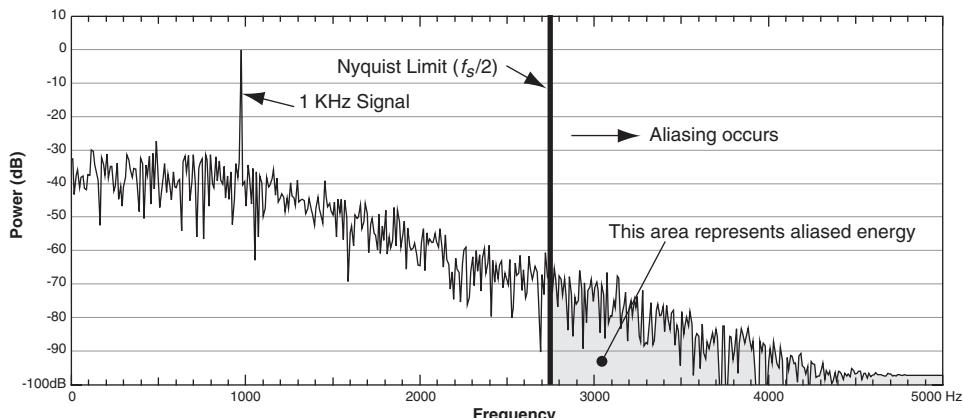
A fundamental rule of sampled data systems is that the input signal must be sampled at a rate greater than twice the highest-frequency component in the signal. This is known as the **Shannon sampling theorem**, and the critical sampling rate is called the **Nyquist rate**.



**Figure 13.1** Graphical display of the effects of sampling rates. When the original 1-kHz sine wave is sampled at 1.2 kHz (too slow), it is totally unrecognizable in the data samples. Sampling at 5.5 kHz yields a much better representation. What would happen if there was a lot of really high-frequency noise?

Stated as a formula, it says that  $f_s/2 > f_a$ , where  $f_s$  is the sampling frequency and  $f_a$  is the maximum frequency of the signal being sampled. Violating the Nyquist criterion is called **undersampling** and results in **aliasing**. Look at Figure 13.1 which simulates a sampled data system. I started out with a simple 1-kHz sine wave (dotted lines), and then I sampled it at two different frequencies, 1.2 and 5.5 kHz. At 5.5 kHz, the signal is safely below the Nyquist rate, which would be 2.75 kHz, and the data points look something like the original (with a little bit of information thrown out, of course). But the data with a 1.2-kHz sampling rate is aliased: It looks as if the signal is 200 Hz, not 1 kHz. This effect is also called **frequency foldback**: Everything above  $f_s/2$  is folded back into the sub- $f_s/2$  range. If you undersample your signal and get stuck with aliasing in your data, can you undo the aliasing? In most cases, no. As a rule, you should not undersample if you hope to make sense of waveform data. Exceptions do occur in certain *controlled* situations. An example is **equivalent-time sampling** in digital oscilloscopes, in which a repetitive waveform is sampled at a low rate, but with careful control of the sampling delay with respect to a precise trigger.

Let's go a step further and consider a nice 1-kHz sine wave, but this time we add some high-frequency noise to it. We already know that a 5.5-kHz sample rate will represent the sine wave all right, but any noise that is beyond  $f_s/2$  (2.75 kHz) will alias. Is this a disaster? That depends on the **power spectrum** (amplitude squared versus frequency) of the noise or interfering signal. Say that the noise is very, very small in amplitude—much less than the resolution of your ADC. In that case it will be undetectable, even though it violates the Nyquist criterion.



**Figure 13.2** Power spectrum of a 1-kHz sine wave with low-pass-filtered noise added. If the ADC resolves 16 bits, its spectral noise floor is  $\approx$ 115 dB. Assuming that we sample at 5.5 kHz, any energy appearing above  $f_s/2$  (2.75 kHz) and above  $\approx$ 115 dB will be aliased.

The real problems are medium-amplitude noise or spurious signals. In Figure 13.2, I simulated a 1-kHz sine wave with low-pass-filtered white noise added to it. If we use a 16-bit ADC, the specifications say its spectral noise floor is about  $-115$  dB below full-scale when displayed as a power spectrum. (This corresponds to an rms signal-to-noise ratio of 98.08 dB for all frequencies up to the Nyquist limit.) Assuming that the signal is sampled at 5.5 kHz as before, the Nyquist limit is 2.75 kHz.

Looking at the power spectrum, you can see that some of the noise power is above the floor for the ADC, and there is also noise present at frequencies above 2.75 kHz. The shaded triangle represents aliased energy and gives you a qualitative feel for how much contamination you can expect. Exactly what the contamination will look like is anybody's guess; it depends on the nature of the out-of-band noise. In this case, you can be pretty sure that none of the aliased energy will be above  $-65$  dB. The good news is that this is just plain old uncorrelated noise, so its aliased version will also be plain old uncorrelated noise. It is not a disaster, just another noise source.

### Filtering and Averaging

To get rid of aliasing problems, you need to use a **low-pass filter**, known in this case as an **antialiasing filter**. Analog filters are mandatory, regardless of the sampling rate, unless you know the signal's frequency characteristics and can live with the aliased noise. There has to be *something* to limit the bandwidth of the raw signal to  $f_s/2$ . The analog filter can be in the transducer, in the signal conditioner,

on the ADC board, or in all three places. One problem with analog filters is that they can become very complex and expensive. If the desired signal is fairly close to the Nyquist limit, the filter needs to cut off very quickly, implying lots of stages (this is more formally known as the **order** of the filter's **transfer function**). High-performance antialiasing filters, such as the SCXI-1141, have such high-order designs and meet the requirements of most situations.

**Digital filters** can augment, but not replace, analog filters. Digital filter VIs are included with the LabVIEW analysis library, and they are functionally equivalent to analog filters. The simplest type of digital filter is a **moving averager** (examples of which are available with LabVIEW) which has the advantage of being usable in real time on a sample-by-sample basis. One way to simplify the antialiasing filter problem is to **oversample** the input. If your ADC hardware is fast enough, just turn the sampling rate way up, and then use a digital filter to eliminate the higher frequencies that are of no interest. This makes the analog filtering problem much simpler because the Nyquist frequency has been raised much higher, so the analog filter doesn't have to be so sharp. A compromise is always necessary: You need to sample at a rate high enough to avoid significant aliasing with a modest analog filter; but sampling at too high a rate may not be practical because the hardware is too expensive and/or the flood of extra data may overload your poor CPU.

A potential problem with averaging arises when you handle nonlinear data. The process of averaging is defined to be the summation of several values, divided by the number of values. If your data is, for instance, exponential, then averaging values (a linear operation) will tend to bias the data. (Consider the fact that  $e^x + e^y$  is not equal to  $e^{(x+y)}$ .) One solution is to linearize the data before averaging. In the case of exponential data, you should take the logarithm first. You may also be able to ignore this problem if the values are closely spaced—small pieces of a curve are effectively linear. It's vital that you understand your signals qualitatively and quantitatively before you apply *any* numerical processing, no matter how innocuous it may seem.

If your main concern is the rejection of 60-Hz line frequency interference, an old trick is to average an array of samples over one line period (16.66 ms in the United States). For instance, you could acquire data at 600 Hz and average groups of 10, 20, 30, and so on, up to 600 samples. You should do this for every channel. Using plug-in boards with LabVIEW's data acquisition drivers permits you to adjust the sampling interval with high precision, making this a reasonable option. Set up a simple experiment to acquire and average data from a noisy input. Vary the sampling period, and see if there isn't a null in the noise level at each 16.66-ms multiple.

If you are attempting to average recurrent waveforms to reduce noise, remember that the arrays of data that you acquire must be perfectly in phase. If a phase shift occurs during acquisition, then your waveforms will partially cancel one another or cause distortion. **Triggered** data acquisition (discussed later) is the normal solution because it helps to guarantee that each buffer of data is acquired at the same part of the signal's cycle.

Some other aspects of filtering that may be important for some of your applications are **impulse response** and **phase response**. For ordinary datalogging, these factors are generally ignored. But if you are doing dynamic testing such as in vibration analysis, acoustics, or seismology, then impulse and phase response can be very important. As a rule of thumb, when filters become very complex (high-order), they cut off more sharply, have more radical phase shifts around the cutoff frequency, and, depending on the filter type, exhibit greater ringing on transients. Overall, filtering is a rather complex topic that is best left to the references; the *Active Filter Cookbook* (Lancaster, 1975) is an old favorite. Even with an electrical engineering degree and lots of experience with filters, Gary still uses it to get a quick answer to practical filtering problems.

The best way to analyze your filtering needs is to use a spectrum analyzer. Then you know exactly what signals are present and what has to be filtered out. You can use a dedicated spectrum analyzer instrument (*very expensive*), a digital oscilloscope with FFT capability (or let LabVIEW do the power spectrum), or even a multifunction I/O board running as fast as possible with LabVIEW doing the power spectrum. Spectrum analyzers are included in the data acquisition examples distributed with LabVIEW, and they work quite well.

## About ADCs, DACs, and Multiplexers

Important characteristics of an ADC or a digital-to-analog converter (DAC) are resolution, range, speed, and sources of error. (For a detailed look at all these parameters, consult the *Analog-Digital Conversion Handbook*, available from Analog Devices.)

**Resolution** is the number of bits that the ADC uses to represent the analog signal. The greater the number of bits, the finer the resolution of the converter. Figure 13.3 demonstrates the resolution of a hypothetical 3-bit converter, which can resolve  $2^3$ , or 8, different levels. The sine wave in this figure is not well represented because of the rather coarse **quantization** levels available. Common ADCs have resolutions of 8, 12, and 16 bits, corresponding to 256, 4096, and 65,536 quantization levels. Using high-resolution converters is generally desirable, although they tend to be somewhat slower.

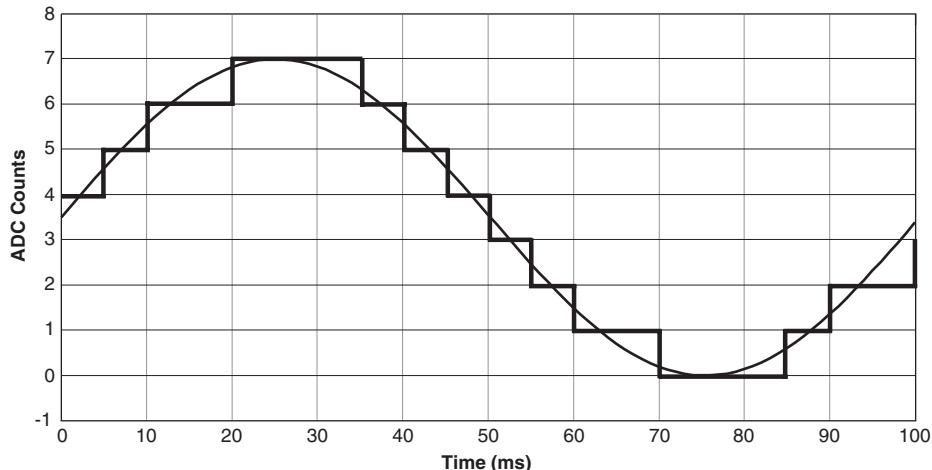


Figure 13.3 A sine wave and its representation by a 3-bit ADC sampling every 5 ms.

**Range** refers to the maximum and minimum voltage levels that the ADC can quantize. Exceeding the input range results in what is variously termed *clipping*, *saturation*, or *overflow/underflow*, where the ADC gets stuck at its largest or smallest output code. The **code width** of an ADC is defined as the change of voltage between two adjacent quantization levels or, as a formula,

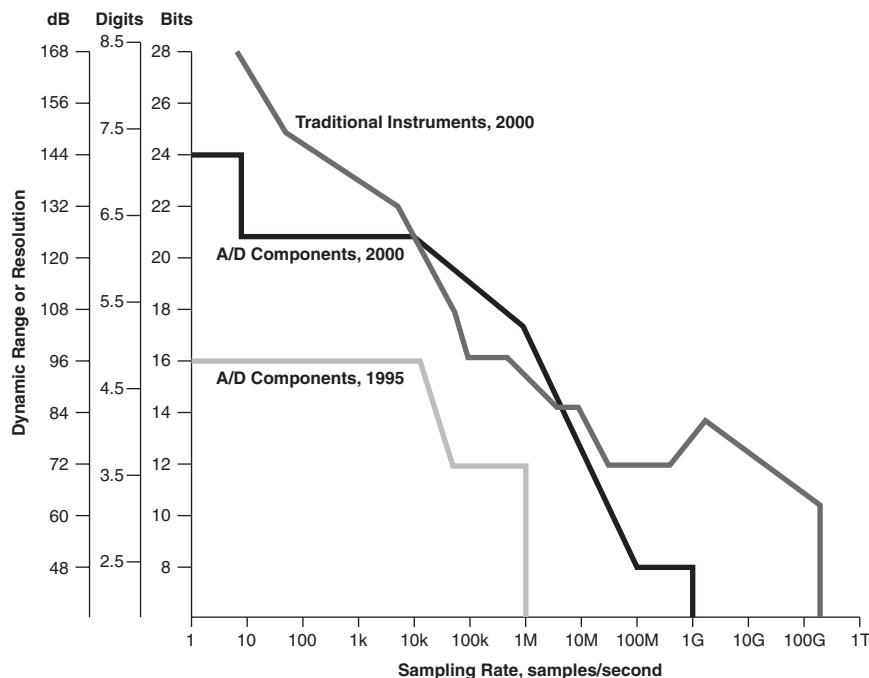
$$\text{Code width} = \frac{\text{range}}{2^N}$$

where  $N$  is the number of bits and code width and range are measured in volts. A high-resolution converter (lots of bits) has a small code width. The intrinsic range, resolution, and code width of an ADC can be modified by preceding it with an amplifier that adds gain. The code width expression then becomes

$$\text{Code width} = \frac{\text{range}}{\text{gain} \times 2^N}$$

High gain thus narrows the code width and enhances the resolution while reducing the effective range. For instance, a common 12-bit ADC with a range of 0 to 10 V has a code width of 2.44 mV. By adding a gain of 100, the code width becomes 24.4  $\mu$ V, but the effective range becomes  $10/100 = 0.1$  V. It is important to note the tradeoff between resolution and range when you change the gain. High gain means that overflow may occur at a much lower voltage.

**Conversion speed** is determined by the technology used in designing the ADC and associated components, particularly the **sample-and-hold (S/H)** amplifier that freezes the analog signal just long enough to do the conversion. Speed is measured in time per conversion or samples



**Figure 13.4** The performance of A/D components—including plug-in boards—now closely approaches or exceeds the performance of rack-and-stack instruments, and usually at much lower cost. (*Courtesy of National Instruments.*)

per second. Figure 13.4 compares the typical resolutions and conversion speeds of plug-in ADC boards and traditional instruments. A very common tradeoff is resolution versus speed; it simply takes more time or is more costly to precisely determine the exact voltage. In fact, high-speed ADCs, such as flash converters that are often used in digital oscilloscopes, decrease in effective resolution as the conversion rate is increased. Your application determines what conversion speed is required.

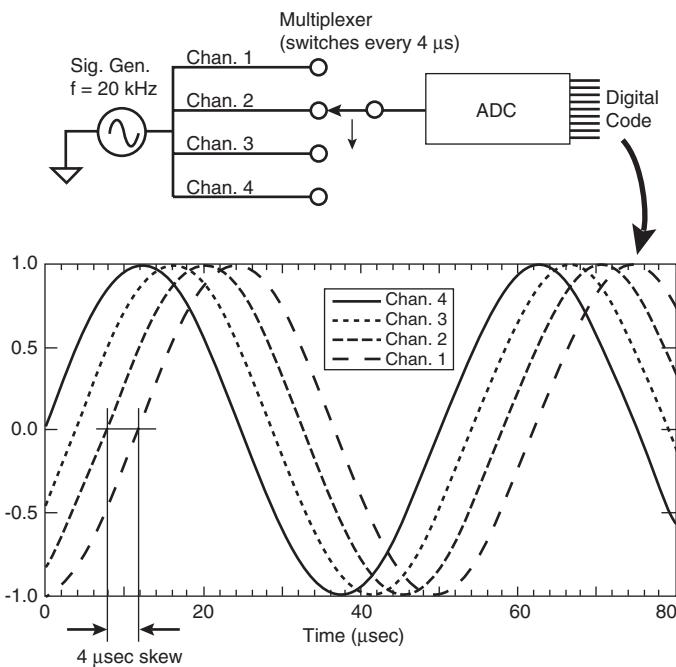
There are many sources of **error** in ADCs, some of which are a little hard to quantify; in fact, if you look at the specification sheets for ADCs from different manufacturers, you may not be able to directly compare the error magnitudes because of the varying techniques the manufacturers may have used.

Simple errors are **gain** and **offset** errors. An ideal ADC follows the equation for a straight line,  $y = mx + b$ , where  $y$  is the input voltage,  $x$  is the output code, and  $m$  is the code width. Gain errors change the slope  $m$  of this equation, which is a change in the code width. Offset errors change the intercept  $b$ , which means that 0-V input doesn't give you zero counts at the output. These errors are easily corrected through calibration. Either you can adjust some trimmer potentiometers so that zero and full-scale match a calibration standard, or you can make

measurements of calibration voltages and fix the data through a simple straight-line expression in software. Fancy ADC systems, such as the E-series plug-in boards from National Instruments, include self-calibration right on the board.

**Linearity** is another problem. Ideally, all the code widths are the same. Real ADCs have some linearity errors that make them deviate from the ideal. One measure of this error is the **differential nonlinearity**, which tells you the worst-case deviation in code width. **Integral nonlinearity** is a measure of the ADC transfer function. It is measured as the worst-case deviation from a straight line drawn through the center of the first and last code widths. An ideal converter would have zero integral nonlinearity. Nonlinearity problems are much more difficult to calibrate out of your system. To do so would mean taking a calibration measurement at each and every quantization level and using that data to correct each value. In practice, you just make sure that the ADC is tightly specified and that the manufacturer delivers the goods as promised.

If a **multiplexer** is used before an ADC to scan many channels, **timing skew** will occur (Figure 13.5). Since the ADC is being switched



**Figure 13.5** Demonstration of skew in an ADC with a multiplexer. Ideally, there would be zero switching time between channels on the multiplexer; this one has 4  $\mu$ s. Since the inputs are all the same signal, the plotted data shows an apparent phase shift.

between several inputs, it is impossible for it to make all the measurements simultaneously. A delay between the conversion of each channel results, and this is called *skew*. If your measurements depend on critical timing (phase matching) between channels, you need to know exactly how much skew there is in your ADC system.

Multiplexers are used as cost-saving devices, since ADCs tend to be among the most expensive parts in the system, but the skew problem can be intolerable in some applications. Then multiple ADCs or sample-and-hold amplifiers (one per channel) are recommended. This is the approach taken by the National Instruments SCXI-1140 and 4450-series dynamic signal acquisition boards, respectively.

You can remove the timing skew in software as long as you know exactly what the skew is. Most plug-in DAQ boards work as follows: By default, the multiplexer scans through the channel list at the top speed of the ADC. For instance, if you have a 100-kHz ADC, there will be a skew of approximately 10  $\mu$ s between channels. However, the hardware and/or the driver software may increase this value somewhat due to all the extra settling time. For best results, you probably will have to acquire some test data to be sure. You can also choose to adjust the **interchannel delay** (i.e., the skew) with the DAQ VIs in LabVIEW. That feature lets *you* specify the skew, which makes it easier to correct the data's time base. But chances are, even with your adjustments and corrections, the results won't be quite as accurate as the multiple-ADC approach, and that's why we prefer to use an optimum hardware solution.

One misconception about ADCs is that they always do some averaging of the input signal between conversions. For instance, if you are sampling every millisecond, you might expect the converted value to represent the average value over the last millisecond of time. This is true only for certain classes of ADCs, namely, dual-slope integrators and voltage-to-frequency converters. Ubiquitous high-speed ADCs, as found on most multifunction boards, are always preceded by sample-and-hold amplifiers.

The S/H samples the incoming signal for an extremely brief time (the **aperture time**) and stores the voltage on a capacitor for measurement by the ADC. The aperture time depends on the resolution of the ADC and is in the nanosecond range for 16-bit converters. Therefore, if a noise spike should occur during the aperture time, you will get an accurate measurement of the *spike*, not the real waveform. This is one more reason for using a low-pass filter.

When a multiplexer changes from one channel to another, there is a period of time during which the output is in transition, known as the **settling time**. Because of the complex nature of the circuitry in these systems, the transition may not be as clean as you might expect. There may be an overshoot with some damped sinusoidal ringing. Since your objective is to obtain an accurate representation of the input signal,

you must wait for these aberrations to decay away. Settling time is the amount of time required for the output voltage to begin tracking the input voltage within a specified error band after a change of channels has occurred. It is clearly specified on all ADC system data sheets. You should not attempt to acquire data faster than the rate determined by the settling time plus the ADC conversion time.

Another unexpected source of input error is sometimes referred to as **charge pump-out**. When a multiplexer switches from one channel to the next, the input capacitance of the multiplexer (and the next circuit element, such as the sample-and-hold) must charge or discharge to match the voltage of the new input signal. The result is a small glitch induced on the input signal, either positive or negative, depending upon the relative magnitude of the voltage on the preceding channel. If the signal lines are long, you may also see ringing. Charge pump-out effects add to the settling time in an unpredictable manner, and they may cause momentary overloading or gross errors in high-impedance sources. This is another reason to use signal conditioning; an input amplifier provides a buffering action to reduce the glitches.

Many systems precede the ADC with a programmable-gain instrumentation amplifier (PGIA). Under software control, you can change the gain to suit the amplitude of the signal on each channel. A true instrumentation amplifier with its inherent differential connections is the predominant type. You may have options, through software registers or hardware jumpers, to defeat the differential mode or use various signal grounding schemes, as on the National Instruments multifunction boards. Study the available configurations and find the one best suited to your application. The one downside to having a PGIA in the signal chain is that it invariably adds some error to the acquisition process in the form of offset voltage drift, gain inaccuracy, noise, and bandwidth. These errors are at their worst at high-gain settings, so study the specifications carefully. An old axiom in analog design is that high gain and high speed are difficult to obtain simultaneously.

### Digital-to-analog converters

Digital-to-analog converters (DACs) perform the reverse action of ADCs: A digital code is scaled to a proportional analog voltage. We use DACs to generate analog stimuli, such as test waveforms and actuator control signals. By and large, they have the same general performance characteristics as do ADCs. Their main limitations, besides the factors we've already discussed, are the **settling time** and **slew rate**. When you make a change in the digital code, the output is expected to change instantaneously to the desired voltage. How fast the change actually occurs (measured in volts per second) is the *slew rate*. Hopefully,

a clean, crisp step will be produced. In actuality, the output may overshoot and ring for awhile or may take a more leisurely, underdamped approach to the final value. This represents the *settling time*. If you are generating high-frequency signals (audio or above), you need faster settling times and slew rates. If you are controlling the current delivered to a heating element, these specifications probably aren't much of a concern.

When a DAC is used for waveform generation, it must be followed by a low-pass filter, called a **reconstruction filter**, that performs an antialiasing function in reverse. Each time the digital-to-analog (D/A) output is updated (at an interval determined by the time base), a step in output voltage is produced. This step contains a theoretically infinite number of harmonic frequencies. For high-quality waveforms, this out-of-band energy must be filtered out by the reconstruction filter. DACs for audio and dynamic signal applications, such as National Instruments' 4450 series dynamic signal I/O boards, include such filters and have a spectrally pure output. Ordinary data acquisition boards generally have no such filtering and will produce lots of spurious energy. If spectral purity and transient fidelity are important in your application, be mindful of this fact.

### Digital codes

The pattern of bits—the digital *word*—used to exchange information with an ADC or DAC may have one of several coding schemes, some of which aren't intuitive. If you ever have to deal directly with the I/O hardware (especially in lower-level driver programs), you will need to study these schemes. If the converter is set up for **unipolar** inputs (all-positive or all-negative analog voltages), the binary coding is straightforward, as in Table 13.1. But to represent both polarities of numbers for a **bipolar** converter, a **sign bit** is needed to indicate the signal's

TABLE 13.1 Straight Binary Coding Scheme for Unipolar, 3-Bit Converter

Decimal equivalent	Decimal fraction of full-scale		
	Positive	Negative	Straight binary
7	$\frac{7}{8}$	$-\frac{7}{8}$	111
6	$\frac{6}{8}$	$-\frac{6}{8}$	110
5	$\frac{5}{8}$	$-\frac{5}{8}$	101
4	$\frac{4}{8}$	$-\frac{4}{8}$	100
3	$\frac{3}{8}$	$-\frac{3}{8}$	011
2	$\frac{2}{8}$	$-\frac{2}{8}$	010
1	$\frac{1}{8}$	$-\frac{1}{8}$	001
0	$\frac{0}{8}$	$-\frac{0}{8}$	000

**TABLE 13.2 Some Commonly Used Coding Schemes for Bipolar Converters, a 4-Bit Example**

Decimal equivalent	Fraction of full-scale	Sign and magnitude	Two's complement	Offset binary
7	7/8	0111	0111	1111
6	6/8	0110	0110	1110
5	5/8	0101	0101	1101
4	4/8	0100	0100	1100
3	3/8	0011	0011	1011
2	2/8	0010	0010	1010
1	1/8	0001	0001	1001
0	0+	0000	0000	1000
0	0-	1000	0000	1000
-1	-1/8	1001	1111	0111
-2	-2/8	1010	1110	0110
-3	-3/8	1011	1101	0101
-4	-4/8	1100	1100	0100
-5	-5/8	1101	1011	0011
-6	-6/8	1110	1010	0010
-7	-7/8	1111	1001	0001
-8	-8/8	Not represented	1000	0000

NOTE: The sign and magnitude scheme has two representations for zero, but can't represent -8. Also, the only difference between offset binary and two's complement is the polarity of the sign bit.

polarity. The bipolar coding schemes shown in Table 13.2 are widely used. Each has advantages, depending on the application.

## Triggering and Timing

**Triggering** refers to any method by which you synchronize an ADC or DAC to some event. If there is a regular event that causes each individual ADC, it's called the **time base**, or *clock*, and it is usually generated by a crystal-controlled clock oscillator. For this discussion, we'll define triggering as an event that starts or stops a series of conversions which are individually paced by a time base.

When should you bother with triggering? One situation is when you are waiting for a transient event to occur—a single pulse. It would be wasteful (or maybe impossible) to run your ADC for a long time, filling up memory and/or disk space, when all you are interested in is a short burst of data before and/or after the trigger event. Another use for triggering is to force your data acquisition to be in phase with the signal. Signal analysis may be simplified if the waveform always starts with the same polarity and level. Or, you may want to acquire many buffers of data from a recurrent waveform (such as a sine wave) in order to average them, thus reducing the noise. Trigger sources come in three flavors: external, internal, or software-generated.

**External** triggers are digital pulses, usually produced by specialized hardware or a signal coming from the equipment that you are

interfacing with. An example is a function generator that has a connector on it called *sync* which produces a TTL pulse every time an output waveform crosses 0 V in the positive direction. Sometimes you have to build your own trigger generator. When you are dealing with pulsed light sources (such as some lasers), an optical detector such as a photodiode can be used to trigger a short but high-speed burst of data acquisition. Signal conditioning is generally required for external triggers because most data acquisition hardware demands a clean pulse with a limited amplitude range. Chapter 19, “Physics Applications,” goes into detail on external triggering.

**Internal triggering** is built into many data acquisition devices, including oscilloscopes, transient recorders, and multifunction boards. It is basically an analog function in which a device called a **comparator** or **discriminator** detects the signal’s crossing of a specified level. The slope of the signal may also be part of the triggering criteria. Really sophisticated instruments permit triggering on specified patterns, which is especially useful in digital logic and communications signal analysis. The onboard triggering features of National Instruments’ boards are easy to use in LabVIEW, courtesy of the data acquisition VIs. Newer boards include an advanced system timing controller chip, the DAQ-STC, with programmable function inputs that solve some of the more difficult triggering problems you may encounter.

**Software-generated triggers** require a program that evaluates an incoming signal or some other status information and decides when to begin saving data or generating an output. A trivial example is the Run button in LabVIEW that starts up a simple data acquisition program. In that case, the operator is the triggering system. On-the-fly signal analysis is a bit more complex and quickly runs into performance problems if you need to look at fast signals. For instance, you may want to save data from a spectrum analyzer only when the process temperature goes above a certain limit. That should be no problem, since the temperature probably doesn’t change very fast. A difficult problem would be to evaluate the distortion of an incoming audio signal and save only the waveforms that are defective. That might require DSP hardware; it might not be practical at all, at least in real time. The NI DAQ driver includes basic software trigger functionality, such as level and slope detection, that works with many plug-in boards and is very easy to use.

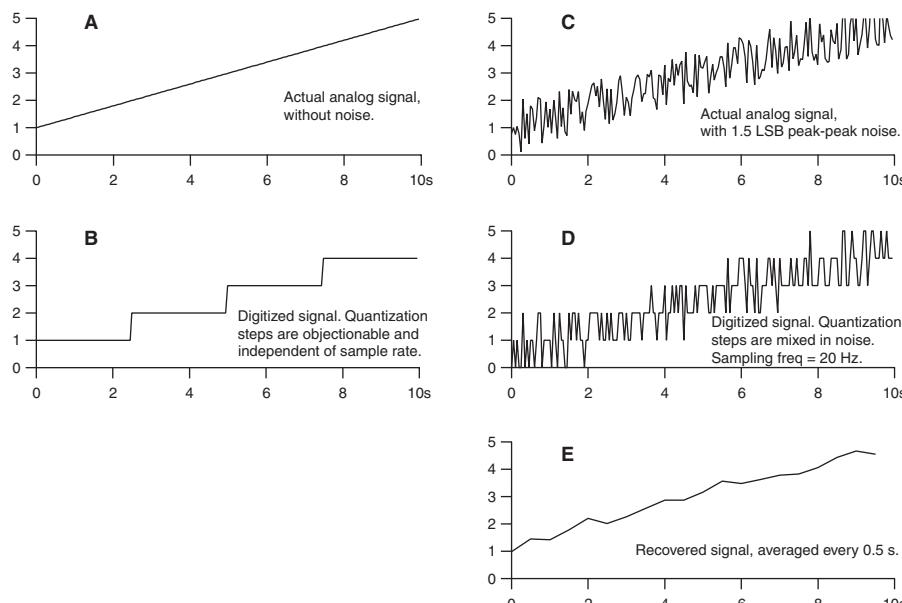
### A Little Noise Can Be a Good Thing

Performance of an analog-to-digital (A/D) or D/A system can be enhanced by adding a small amount of noise and by averaging (Lipshitz, Wanna-maker, and Vanderkooy 1992). Any time the input voltage is somewhere

between two quantization levels of the A/D system and there is some **dither** noise present, the least-significant bit (LSB) tends to toggle among a few codes. For instance, the duty cycle of this toggling action is exactly 50 percent if the voltage is exactly between the two quantization levels. Duty cycle and input voltage track each other in a nice, proportional manner (except if the converter demonstrates some kind of nonlinear behavior). All you have to do is to filter out the noise, which can be accomplished by averaging or other forms of digital filtering.

A source of uncorrelated dither noise, about 1 LSB peak to peak or greater, is required to make this technique work. Some high-performance ADC and DAC systems include dither noise generators; digital audio systems and the National Instruments dynamic signal acquisition boards are examples. High-resolution converters (16 bits and greater) generally have enough thermal noise present to supply the necessary dithering. Incidentally, this resolution enhancement occurs even if you don't apply a filter; filtering simply reduces the noise level.

Figure 13.6 demonstrates the effect of dither noise on the quantization of a slow, low-amplitude ramp signal. To make this realistic, say that the total change in voltage is only about 4 LSBs over a period of 10 s (Figure 13.6A). The vertical axis is scaled in LSBs for clarity.



**Figure 13.6** Graphs in (A) and (B) represent digitization of an ideal analog ramp of 4-LSB amplitude which results in objectionable quantization steps. Adding 1-LSB peak-peak dither noise and low-pass filtering [graphs in (C) through (E)] improves results.

The sampling rate is 20 Hz. In Figure 13.6B, you can see the coarse quantization steps expected from an ideal noise-free ADC. Much imagination is required to see a smooth ramp in this graph. In Figure 13.6C, dither noise with an amplitude of 1.5 LSB peak to peak has been added to the analog ramp. Figure 13.6D is the raw digitized version of this noisy ramp. Contrast this with Figure 13.6B, the no-noise case with its coarse steps. To eliminate the random noise, we applied a simple *boxcar* filter where every 10 samples (0.5 s worth of data) is averaged into a single value; more elaborate digital filtering might improve the result. Figure 13.6E is the recovered signal. Clearly, this is an improvement over the ideal, noiseless case, and it is very easy to implement in LabVIEW. In Chapter 14, “Writing a Data Acquisition Program,” we’ll show you how to oversample and average to improve your measurements.

Low-frequency analog signals give you some opportunities to further improve the quality of your acquired data. At first glance, that thermocouple signal with a sub-1-Hz bandwidth and little noise could be adequately sampled at 2 or 3 Hz. But by **oversampling**—sampling at a rate several times higher than the minimum specified by the Nyquist rate—you can enhance resolution and noise rejection.

Noise is reduced in proportion to the square root of the number of samples that are averaged. For example, if you average 100 samples, the standard deviation of the average value will be reduced by a factor of 10 when compared to a single measurement. Another way of expressing this result is that you get a 20-dB improvement in signal-to-noise ratio when you average 100 times as many samples. This condition is true as long as the A/D converter has good linearity and a small amount of dither noise. This same improvement occurs with repetitive signals, such as our 1-kHz sine wave with dither noise. If you synchronize your ADC with the waveform by triggering, you can average several waveforms. Since the noise is not correlated with the signal, the noise once again averages to zero according to the square root rule. How you do this in LabVIEW is discussed in Chapter 14, “Writing a Data Acquisition Program.”

## Throughput

A final consideration in your choice of converters is the **throughput** of the system, a yardstick for overall performance, usually measured in samples per second. Major factors determining throughput are as follows:

- A/D or D/A conversion speed
- Use of multiplexers and amplifiers, which may add delays between channels

- Disk system performance, if streaming data to or from disk
- Use of direct memory access (DMA), which speeds data transfer
- CPU speed, especially if a lot of data processing is required
- Operating system overhead

The glossy brochure or data sheet you get with your I/O hardware rarely addresses these very real system-oriented limitations. Maximum performance is achieved when the controlling program is written in assembly language, one channel is being sampled with the amplifier gain at minimum, and data is being stored in memory with no analysis or display of any kind. Your application will always be somewhat removed from this particular benchmark.

Practical disk systems have many throughput limitations. The disk itself takes a while to move the recording heads around and can only transfer so many bytes per second. The file system and any data conversion you have to do in LabVIEW are added as overhead. If everything is working well, a stream-to-disk LabVIEW VI will continuously run in excess of 6 Mbytes/s. For really fast I/O, you simply have to live within the limits of available memory. But memory is cheap these days, and the performance is much better than that of any disk system. If you really need 1 Gbyte of RAM to perform your experiment, then don't fool around, *just buy it*. Tell the purchasing manager that Gary said so.

**Double-buffered DMA** for acquisition and waveform generation is built into the LabVIEW support for many I/O boards and offers many advantages in speed because the hardware does all the real-time work of transferring data from the I/O to main memory. Your program can perform analysis, display, and archiving tasks while the I/O is in progress (you can't do *too* much processing though). Refer to your LabVIEW data acquisition VI library reference manual for details on this technique.

Performance without DMA is quite limited. Each time you send a command to a plug-in board, the command is processed by the NI DAQ driver, which in turn must get permission from the operating system to perform an I/O operation. There is a great deal of overhead in this process, typically on the order of a few hundred microseconds. That means you can read one sample at a time from an input at a few kilohertz without DMA. Clearly, this technique is not very efficient and should be used only for infrequent I/O operations.

If you need to do very much on-the-fly analysis, adding a **DSP board** can augment the power of your computer's CPU by off-loading tasks such as FFT computations. Using a DSP board as a general-purpose computer is another story—actually another *book!* Programming such a machine to orchestrate data transfers, do control algorithms, and so on generally requires programming in C or assembly language, using

the support tools for your particular DSP board. If you are an experienced programmer, this is a high-performance alternative.

Access to external DSP horsepower without low-level programming is available from **Sheldon Instruments** with its LabVIEW add-on product called **QuVIEW**. It works with Sheldon Instruments' PCI-based DSP boards that feature AT&T's 32C or TI's C3x processors coupled to multifunction I/O modules. With QuVIEW, you use a collection of VIs that automatically download code to the DSP memory for fast execution independent of the host PC. It's especially good for continuous algorithms such as signal generation, filtering, FFTs, and streaming data to disk. If the extensive library of built-in functions isn't enough, you can create your own external code, using C compilers or assemblers from TI or Tartan, or graphical DSP code generators from MathWorks or Hyperception. In either case, your external code is neatly encapsulated in familiar VI wrappers for easy integration into a LabVIEW application.

## Bibliography

- 3M Specialty Optical Fibers: *Fiber Optic Current Sensor Module* (product information and application note), West Haven, Conn., 1996
- Beckwith, Thomas G., and R. D. Marangoni: *Mechanical Measurements*, Addison-Wesley, Reading, Mass., 1990.
- Gunn, Ronald: "Designing System Grounds and Signal Returns," *Control Engineering*, May 1987.
- Lancaster, Donald: *Active Filter Cookbook*, Howard W. Sams & Co., Indianapolis, Ind., 1975.
- Lipshitz, Stanley P., R. A. Wannamaker, and J. Vanderkooy: "Quantization and Dither: A Theoretical Survey," *J. Audio Eng. Soc.*, vol. 40, no. 5, 1992, pp. 355-375.
- Morrison, Ralph: *Grounding and Shielding Techniques in Instrumentation*, Wiley-Interscience, New York, 1986.
- Norton, Harry R.: *Electronic Analysis Instruments*, Prentice-Hall, Englewood Cliffs, N.J., 1992.
- Omega Engineering, Inc.: *Temperature Handbook*, Stamford, Conn., 2000. (Available free by calling 203-359-1660 or 800-222-2665.)
- Ott, Henry W.: *Noise Reduction Techniques in Electronic Systems*, Wiley, New York, 1988.
- Pallas-Areny, Ramon, and J. G. Webster: *Sensors and Signal Conditioning*, Wiley, New York, 1991.
- Qian, Shie, and Dapang Chen: *Joint Time-Frequency Analysis—Methods and Applications*, Prentice-Hall, Englewood Cliffs, N.J., 1996. (Call Prentice-Hall at 800-947-7700 or 201-767-4990.)
- Sheingold, Daniel H.: *Analog-Digital Conversion Handbook*, Prentice-Hall, Englewood Cliffs, N.J., 1986.
- Steer, Robert W., Jr.: "Anti-aliasing Filters Reduce Errors in ADC Converters," *Electronic Design News* (EDN), March 30, 1989.
- White, Donald R. J.: *Shielding Design Methodology and Procedures*, Interference Control Technologies, Gainesville, Va., 1986.

*This page intentionally left blank*

## Writing a Data Acquisition Program

We're going to bet that your first LabVIEW application was (or will be) some kind of data acquisition system. We say that because data acquisition is by far the most common LabVIEW application. Every experiment or process has signals to be measured, monitored, analyzed, and logged, and each signal has its own special requirements. Although it's impossible to design a universal data acquisition system to fit every situation, there are plenty of common architectures that you can use, each containing elements that you can incorporate into your own problem-solving toolkit.

As we mentioned in the Introduction, what you'll learn in this chapter are all the *other* things that the LabVIEW manuals and examples don't cover. The act of fetching data from an input device is the easy part, and it's a subject that is already well discussed. On the other hand, how do you keep track of channel assignments and other configuration information? And how does data analysis affect program design? These topics are important, yet rarely mentioned. It's time to change all that.

Your data acquisition application might include some control (output) functionality as well. Most experiments have some things that need to be manipulated—some valves, a power supply set point, or maybe a motor. That should be no problem as long as you spend some time designing your program with the expectation that you will need some control features. If your situation requires a great deal of control functionality, read Chapter 18, "Process Control Applications," to see some methods that may work better than simply adding on to a data acquisition design.

Plan your application as you would any other, going through the recommended steps. We've added a few items to the procedure that

emphasize special considerations for data acquisition: **data analysis**, **throughput**, and **configuration management** requirements. Here are the basic steps.

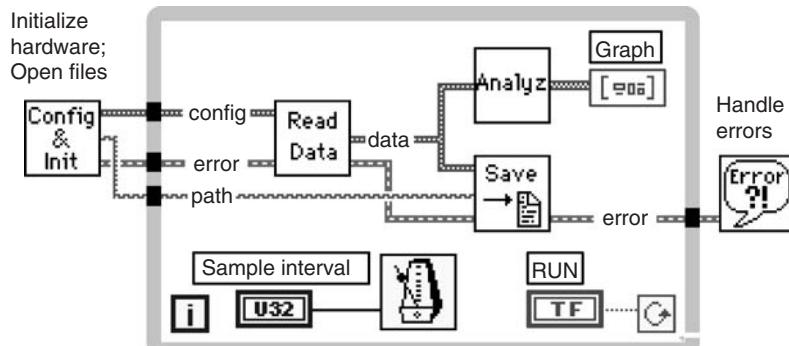
1. Define and understand the problem; define the signals and determine what the data analysis needs are.
2. Specify the type of I/O hardware you will need, and then determine sample rates and total throughput.
3. Prototype the user interface and decide how to manage configurations.
4. Design and then write the program.

If your system requires extra versatility, such as the ability to quickly change channel assignments or types of I/O hardware, then you will need to include features to manage the system's **configuration**. Users should be able to access a few simple controls rather than having to edit the diagram when a configuration change is needed.

Data **throughput**, the aggregate sampling rate measured in samples per second, plays a dominant role in determining the architecture of your program. High sampling rates can severely limit your ability to do real-time analysis and graphics. Even low-speed systems can be problematic when you require accurate timing. Fortunately, there are plenty of hardware and software solutions available in the LabVIEW world.

The reason for assembling a data acquisition system is to **acquire** data, and the reason for acquiring data is to **analyze** it. Surprisingly, these facts are often overlooked. Planning to include analysis features and appropriate file formats will save you (and the recipients of your data) a lot of grief.

The canonical LabVIEW VI for a simple, yet complete, data acquisition program is shown in Figure 14.1. It begins with a VI that handles I/O configuration and the opening of any data files. The rest of the program resides in a While Loop that cycles at a rate determined by the sample interval control. The Read Data VI communicates with hardware and returns the raw data, which is then analyzed for display and stored in files. All subVIs that can produce an error condition are linked by an error I/O cluster, and an error handler tells the user what went wrong. Simple as this diagram is, it could actually work, and do some rather sophisticated processing at that. Try simplifying your next data acquisition problem to this level. Add functionality as required, but keep things modular so that it's easy to understand and modify. This chapter is devoted to the building blocks shown in Figure 14.1.



**Figure 14.1** A generic data acquisition program includes the functions shown here.

## Data Analysis and Storage

**Data analysis** has a different meaning in every application. It depends on the kind of signals you are faced with. For many applications, it means calculating simple statistics (minimum, maximum, mean, standard deviation, etc.) over some period of time. In spectroscopy and chromatography, it means peak detection, curve fitting, and integration. In acoustics and vibration studies, it means Fourier transforms, filtering, and correlation. Each type of analysis affects your LabVIEW program design in some way. For instance, doing a fast Fourier transform (FFT) on a large array in real time requires lots of processing power—your system could become so burdened that data collection might be disrupted. Such analysis drives the performance requirements of your VIs. And everyone worries about timing information, both for real-time analysis and for reading data from files. It's obvious that your program has to measure and store time markers reliably and in a format that is useful to the analysis programs.

What you need to avoid is *analysis by accident*. Time and again we've seen LabVIEW programs that grab data from the hardware and stuff it into a file with no thought about compatibility with the analysis program. Then the poor analyst has to grind along, parsing the file into readable pieces, trying to reconstitute important features of the data set. Sometimes, the important information isn't available on disk at all, and you *hope* that it has been written down *somewhere*. Disaster! Gastric distress also occurs when a new real-time analysis need crops up and your program is so inflexible that the new features can't be added without major surgery.

We recommend a preemptive strike. When someone proposes a new data acquisition system, make it a point to force that person to describe,

in detail, how the data will be analyzed. Make sure he or she understands the implications of storing the megabytes or gigabytes of data that an automated data acquisition system may collect. If there is a collective shrug of shoulders, ask them point-blank, “... Then why are we collecting data at all?” *Do not write your data acquisition program until you understand the analysis requirements.*

Finally, you can get started. Divide the analysis job into real-time and postrun tasks, and determine how each aspect will affect your program.

### Postrun analysis

You can analyze data with LabVIEW, another application, or a custom program written in some other language. Sometimes, more than one analysis program will have to read the same data file. In all cases, you need to decide on a suitable **data file format** that your data acquisition program has to write. The file type (typically text or binary), the structure of the data, and the inclusion of timing and configuration information are all important. If other people are involved in the analysis process, get them involved early in the design process. Write down clear file format specifications. Plan to generate sample data files and do plenty of testing so as to assure everyone that the real data will transfer without problems.

It’s a good idea to structure your program so that a single *data saver* VI is responsible for writing a given data file type. Raw data and configuration information go in, and data files go out. You can easily test this data saver module as a stand-alone VI or call it from a test program before your final application is completed. The result is a module with a clear purpose that is reliable and reusable. LabVIEW can read and write any file format (refer to Chapter 7, “Files,” for a general discussion of file I/O). Which data format you use depends on the program that has to read it.

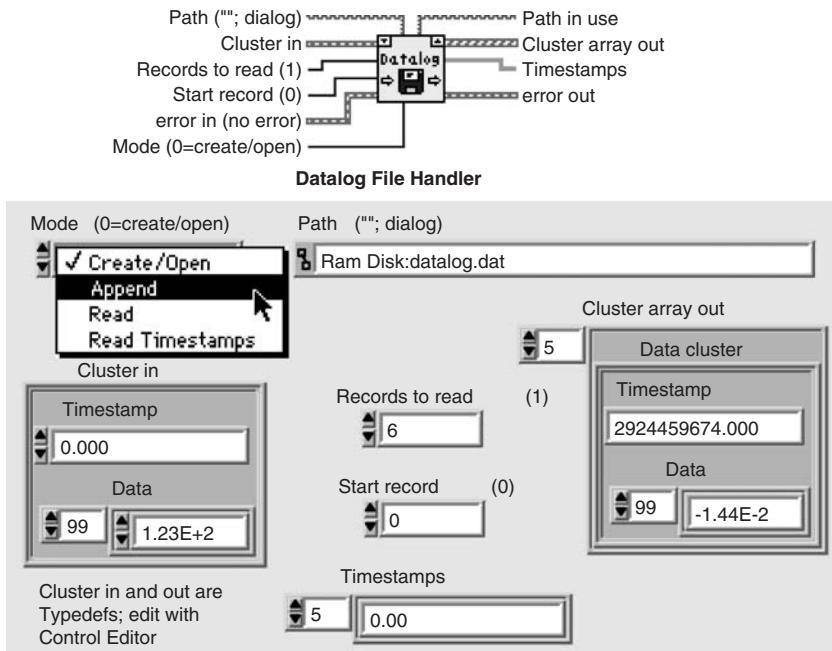
**Datalog file format.** If you plan to analyze data only in LabVIEW, the easiest and most compact format is the **datalog file**, discussed in detail in Chapter 7. A datalog file contains a sequence of binary data **records**. All records in a given file are of the same type, but a record can be a complex data structure, for instance, a cluster containing strings and arrays. The record type is determined when you create the file. You can read records one at a time in a random-access fashion or read several at once, in which case they are returned as an array. This gives your analysis program the ability to use the data file as a simple database, searching for desired records based on one or more key fields in each record, such as a timestamp.

The disadvantage of datalog format files is that they can only be read by LabVIEW or by a custom-written program. However, you can easily write a translator in LabVIEW that reads your datalog format and writes out files with another format. Another hazard (common to all binary file formats) is that you must know the data type used when the file was written; otherwise, you may never be able to decipher the file.

You might be able to use the automatic front panel datalogging features of LabVIEW. They are very easy to use. All you have to do is to turn on the datalogging by using the **Datalogging** submenu in the Operate menu for a subVI that displays the data you wish to save. Every time that the subVI finishes executing (even if its front panel is not displayed), the front panel data are appended to the current log file. The first time the subVI is called, you will receive a dialog asking for a new datalog file. You can also open the subVI and change log files through the Datalogging menu. To access logged data, you can use the file I/O functions, or you can place the subVI of interest in a new diagram and choose **Enable Database Access** from its pop-up menu. You can then read the datalog records one at a time. All the front panel controls are available—they are in a cluster that is conveniently accessed by Unbundle By Name. But for maximum file performance, there's nothing better than wiring the file I/O functions directly into your diagram. Open the datalog file at the start of the experiment, and don't close it until you're done.

**Datalog File Handler VI makes it easy.** It seems as if your diagrams always end up with a large number of file I/O functions scattered all over the place, with all the attendant wiring. One way to clean up the situation is to encapsulate all file I/O operations into a single **integrated VI**. This approach works for many other applications as well, such as instrument drivers. The principle is simple: Create a VI with a **Mode** control wired to a Case structure containing the various operations you need to perform. For files, the modes might be Create, Write, and Read. Gary did exactly that for the **Datalog File Handler VI**, and it really makes datalog files easy to use (Johnson 1994).

On the panel in Figure 14.2, you can see the **Mode** control and its four possible choices. You begin by creating a new file or opening an existing one. The file's path is saved in an uninitialized shift register, so you don't have to wire to the path input for other I/O operations. *Append* mode appends data in the **Cluster in** control to the file. *Read* returns a selected range of data as a cluster array. You can choose the starting record number and the number of records to read. The last mode, *Read Timestamps*, returns an array of timestamp values for the entire file. This tells you the number of records in the file (equal to the number of elements in the **Timestamps** array). It's also a convenient



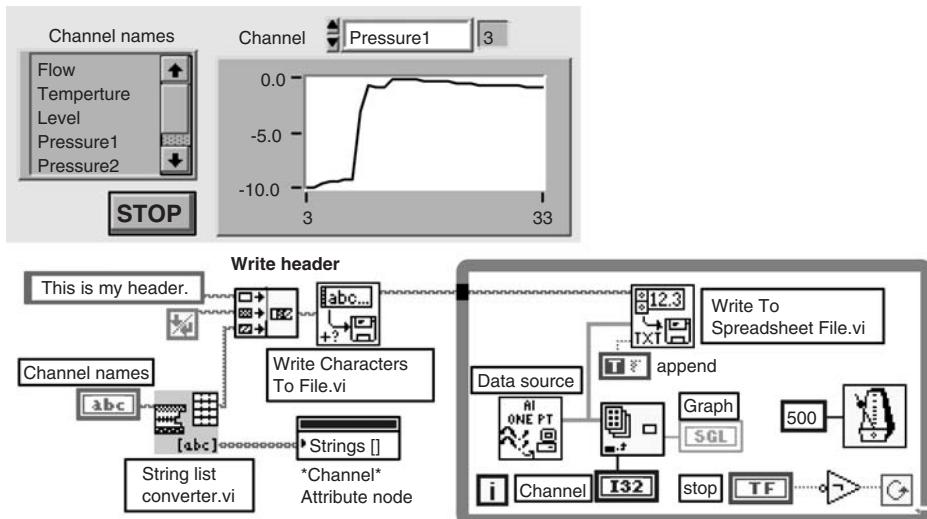
**Figure 14.2** The Datalog File Handler VI encapsulates datalog file I/O operations into a single, integrated VI.

way to locate a particular record of data based on time. The **Cluster in** and **Cluster array out** controls are typeDefs. Modify them with the Control Editor to match your data type. *Note:* The Timestamp is a required item in the cluster.

**ASCII text format.** Good old **ASCII text** files are your best bet for portable data files. Almost every application can load data from a text file that has simple formatting. The ubiquitous **tab-delimited text** format is a likely choice. Format your data values as strings with a tab character between each value, and place a carriage return at the end of the line; then write it out. The only other thing you need to determine is the type of header information. Simple graphing and spreadsheet applications are happy with column names as the first line in the file:

Date	Time	Channel_1	Channel_2
5-13-82	01:17:34	5.678	-13.43
5-13-82	01:17:44	5.665	-13.58

A spreadsheet could be programmed to interpret all kinds of header information, if you need to include more. Other applications are less versatile with respect to headers, so be sure you know who's going to



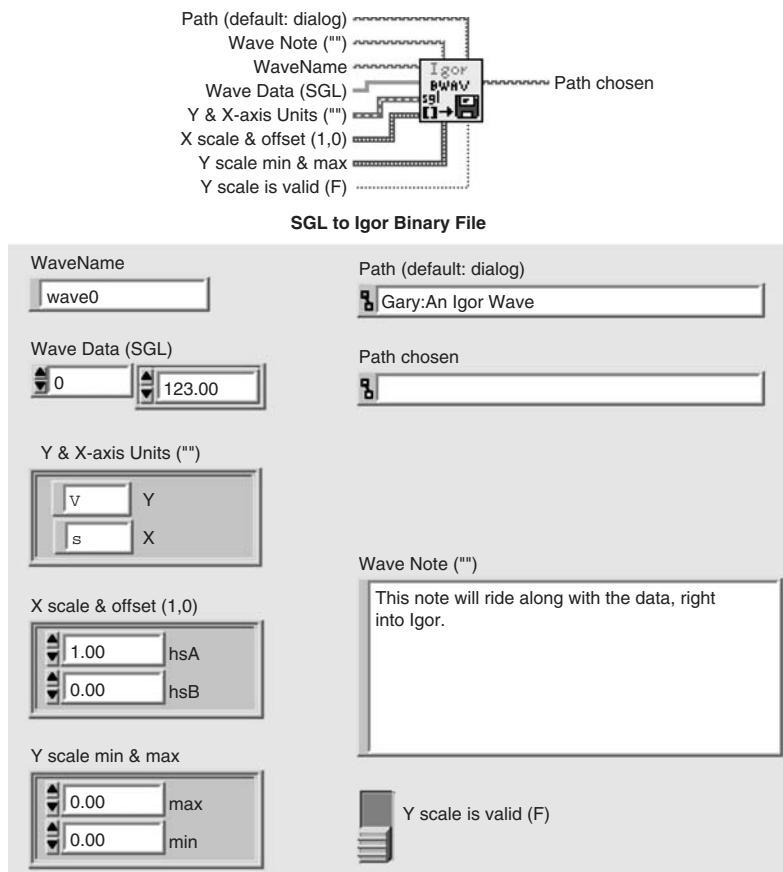
**Figure 14.3** This simple datalogger example writes a text header and then tab-delimited data. The String List Converter VI initializes the Channels control with all the channel names.

be reading your file. Figure 14.3 shows a simple datalogger that writes a text header, followed by tab-delimited text data. It uses the easy-level file VIs for simplicity. We wrote a little utility VI, **String List Converter**, to make it easier for the user to enter channel names. The names are typed into a string control, separated by carriage returns. String List Converter translates the list into a set of tab-delimited names for the file header and into a string array that you can wire to the Strings[] attribute for a ring control, as shown.

The disadvantages of ASCII text files are that they are bulkier than binary files, and they take much longer to read and write (often *several hundred times* longer) because each value has to be converted to and from strings of characters. For high-speed data-recording applications, text files are out of the question. You *might* be able to store a few thousand samples per second as text on a fast computer, but be sure to benchmark carefully before committing yourself to text files.

**Custom binary formats.** LabVIEW can write files with arbitrary binary formats to suit other applications. If you can handle the requisite programming, binary files are really worthwhile—high on performance and very compact. It's also nice to open a binary file with an analysis program and have it load without any special translation. Keep in mind the fact that LabVIEW datalog files are also binary format (and fast, too), but are significantly easier to use, at least within LabVIEW. If you don't really need a custom binary format, stick with datalogs for simplicity.

Binary file handlers require significant programming experience. Even if you have all the formatting information, be prepared to spend time working out the programming details. Software manufacturers will generally supply you with a description of their application's native binary file format if you ask the right person. Usually, you will get some kind of program listing that was lifted from the manufacturer's file I/O routines. If the company is interested in making its file format public, it will supply an application note and sometimes even machine-readable code. The folks at **Wavemetrics** supply all this information for **Igor** (their analysis and graphing package for Macintosh and Windows) with the application. Because the information was available, the VIs to read and write Igor binary were easy to create and are now public domain. Figure 14.4 shows the panel for the VI that writes a



**Figure 14.4** This VI creates a file suitable for direct loading with Wavemetrics' Igor.

single-precision floating-point array to an Igor binary file. The diagram is very complex, so it's not shown. You can obtain these Igor support VIs from Wavemetrics. By the way, Igor can also read arbitrary binary formats directly. With that capability, you can write your LabVIEW data as simple binary arrays and then load them right into Igor with no special programming whatsoever.

Some binary formats such as the previously described Igor format are unsuitable for continuous data acquisition—where you would like to append records one at a time as the experiment proceeds. They are designed for single-shot experiments in which the entire file is written at once. This is fine for single buffers of data from an oscilloscope, but less useful for a simple datalogger. Buffering data in memory (in arrays) is one way to solve this problem, but if your computer should crash, that data might be lost. You could rewrite the entire file occasionally to avoid such a catastrophe, if you are forced to use one of these single-shot formats. On the other hand, if you are *not* stuck with such a predefined format, it is easy and efficient to write continuously to a binary file, as you can see by examining the techniques used in the DAQ disk streaming example VIs.

**Direct links to other applications.** Several other important analysis applications are available with LabVIEW links. The ability to write data directly to another application is very appealing since it makes background analysis without user intervention a reality; it also eliminates the intermediate file conversion hassle.

Starting with LabVIEW 8.0, you can use a **MathScript Script node** to directly execute MathScript scripts from the LabVIEW diagram. It's located in the Programming >> Structures palette and looks and behaves much as a Formula node. You type in any valid MathScript script and assign variables to input and output terminals that you create.

**MATLAB** is another popular multiplatform mathematics and simulation application, with a variety of toolboxes for specialties in science and engineering. While it will certainly read data files (text or binary), you can also use a **MATLAB Script node**, which is very similar to the MathScript Script node, and, of course, only runs under Windows. Remember that you do need a legal copy of MATLAB installed on your machine, but not for the MathScript scripts.

**Microsoft Excel** is very popular for data analysis, with its large installed base on both Windows and Macintosh. Perhaps its major limitations are a lack of a binary file import capability and a maximum of 65,536 rows, making it unsuitable for very large data sets. But there is excellent support for direct interaction with Excel from LabVIEW.

Macintosh users can exchange data via AppleEvents; there are example VIs that read and write data. Windows users can implement DDE or Active X connections. See the LabVIEW examples. Using ActiveX, you can make Excel do just about anything, from creating and saving workbooks to executing macros and reading or writing data. Again, the LabVIEW examples will get you started.

**Timestamps.** Most data that we collect is a function of time. Therefore, timing information needs to be stored along with the data in your files. The precision or resolution of these timestamps will be driven by the requirements of your experiment and the limitations of LabVIEW's timing functions. Requirements for formatting of the timestamps will be determined by the application that reads the data file.

Consider an ordinary datalogging application where a few dozen channels are stored to disk as ASCII text every few seconds. A resolution of 1 s is probably adequate, and a source for this timing information is **Get Date/Time In Seconds**, or one of the other built-in timing functions. You can then format the returned value as seconds relative to 1 Jan 1904, as seconds relative to the start of your experiment, or divide it by 60 to obtain minutes, and so forth. Saving a simple numeric timestamp has the advantage that it is easy to interpret. If you use a spreadsheet application that can manipulate time and date formats (such as mm/dd/yy hh:mm:ss), then **Get Date/Time String** may be appropriate.

You can get similar results by using the simple **Tick Count (ms)** function, which returns a relative number of milliseconds. The uncertainty of these higher-resolution timers depends on your machine and on LabVIEW's workload. You may want to do some testing if you need really accurate software-derived timestamps.

Data that is acquired at a constant rate (periodic sampling) needs a timestamp only at the beginning of the collection period because the time at any sample is easily calculated. This is certainly true of data acquired with hardware-timed I/O using the DAQ library. There is no reason to store a timestamp with every sample if you already know the sample interval with acceptable precision; it would just take up extra disk space. Only data that is taken aperiodically requires such detailed timestamp information.

Sampling rates higher than about 10 Hz usually are handled by data acquisition hardware (or smart I/O subsystems) because there is too much timing uncertainty in fast LabVIEW loops on general-purpose computers. Using hardware can simplify your timestamp requirements. Because you know that the hardware is sampling at a steady rate, there is no need to store a timestamp for every sample period. Rather, you need only save an initial time and a value for the sampling interval.

The data analysis program should then be able to reconstruct the time base from this simple scheme.

A technique used in *really* fast diagnostics is to add a timing **fiducial** pulse to one or more data channels. Also known as a *fid*, this pulse occurs at some critical time during the experiment and is recorded on all systems (and maybe on all channels as well). It's much the same as the room full of soldiers and the commander says, "Synchronize watches." For example, when you are testing explosives, a fiducial pulse is distributed to all the diagnostic systems just before detonation. For analog data channels, the fiducial pulse can be coupled to each channel through a small capacitor, creating a small *glitch* in the data at the critical moment. You can even synchronize nonelectronic systems by generating a suitable stimulus, such as flashing a strobe in front of a movie or video camera. Fiducial pulses are worth considering any time you need absolute synchronization among disparate systems.

If you need accurate time-of-day information, be sure to reset the computer clock before the experiment begins. Personal computer clocks are notorious for their long-term drift. If you are connected to the Internet, you can install a utility that will automatically reset your system clock to a standard time server. For Windows, there are several public domain utilities, such as *WNSTIME* (available from sunsite.unc.edu), that you can install. For Macintosh, you can use the **Network Time Server** option in the Date and Time Control Panel. These utilities even take into consideration the time zone and the daylight savings time settings for your machine. The only other trick is to find a suitable Network Time Protocol server. One we've had good luck with is NASA's [norad.arc.nasa.gov](http://norad.arc.nasa.gov) server.

In Chapter 5, "Timing," we list a host of precision timing devices that you might consider, such as GPS and IRIG standards. Such hardware can provide absolute timing standards with accuracy as good as a few hundred nanoseconds. That's about as good as it gets.

**Passing along configuration information.** Your analysis program may need information about the configuration of the experiment or software that generated the data. In many cases, channel names are all that is needed, and you can pass them along as the column titles in a spreadsheet file. Beyond that, you have two basic choices: Use a separate **configuration file** or add a **file header** to each data file. Both methods are highly dependent upon the ability of the analysis program to read and interpret the information.

If you're using the **Measurement and Automation Explorer (MAX)**, it has a useful feature called *Create Report* (*File >> Create Report*). *Create Report* allows you to save the contents of your DAQmx tasks in HTML format. The information includes channel names

and descriptions, scale factors, hardware device information, and sensor information. This is great information to include with your documentation. It even includes connection diagrams showing you how to connect the signals to your DAQ equipment.

Binary files almost always have headers because the program that reads them needs information about the type and location of the data within. Here is a list of the kind of information contained in a binary file header:

- Experiment identification
- Channel name
- Creation time and date
- Data type (single- or double-precision)
- Data file version (to avoid incompatibility with future versions)
- The y-axis and x-axis units
- The number of data points
- The y-axis and x-axis scale factors
- Flag to indicate whether user comments follow the data segment

As always, the format of this binary file header will be highly specified on a byte-by-byte basis. You need to make sure that each item is of the proper data type and length before writing the header out to the file. An effective way to do this is to assemble all the items into a cluster, **Type Cast** it (see Chapter 4, “LabVIEW Data Types”) to a string, and then write it out, using byte stream mode with the **Write File** function. Alternatively, the header can be text with flag characters or end-of-line characters to help the reading application parse the information.

For text files, generating a header is as simple as writing out a series of strings that have been formatted to contain the desired information. Reading and *decoding* a text-format header, on the other hand, can be quite challenging for any program. If you simply want an experimental record or free-form notepad header for the purposes of documentation, that’s no problem. But parsing information out of the header for programmatic use requires careful design of the header’s format. Many graphing and analysis programs can do little more than read blocks of text into a long string for display purposes; they have little or no capacity for parsing the string. Spreadsheets (and of course programming languages) can search for patterns, extract numbers from strings, and so forth, but not if the format is poorly defined. Therefore, you need to work on both ends of the data analysis problem—reading as well as writing—to make sure that things will play together.

Another solution to this header problem is to use an **index file** that is separate from the data file. The index file contains all the information necessary to successfully load the data including pointers into the data file.

It can also contain configuration information. The data file can be binary or ASCII format, containing only the data values. We've used this technique on several projects, and it adds some versatility. If the index file is ASCII text, then you can print it out to see what's in the data file. Also, the data file may be more easily loaded into programs that would otherwise choke on header information. You still have the problem of loading the configuration, but at least the data can be loaded and the configuration is safely stored on disk. One caution: Don't lose one of the files!

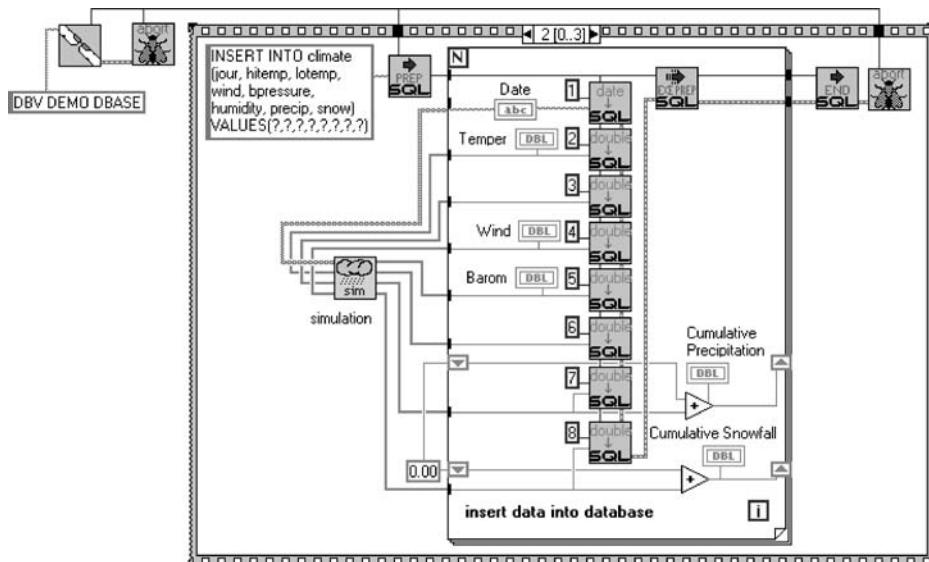
The configuration file can be formatted for direct import into a spreadsheet and used as a printable record of the experiment. This turns out to be quite useful. Try to produce a complete description of the hardware setup used for a given test, including module types, channel assignments, gain settings, and so forth. Here's what a simple configuration file might look like:

```
Source expt RUN306 08-SEP-1992 21:18:51.00 HD:Data:RUN306
File    Module ID   Slot  Crate  Signal Name   Chan  Units
TEST0.T Kinetic 3525 1      1       TC1          1     Deg C
TEST0.T Kinetic 3525 1      1       TC2          2     Deg C
TEST0.T Kinetic 3525 1      1       Upper src temp 4     Deg C
TEST0.T Kinetic 3525 1      1       Lower src temp 5     Deg C
END
```

When the experimenter has a question regarding the signal connections, you can refer to this list, which is usually clipped into the laboratory logbook. We'll discuss some methods for generating configuration files later in this chapter.

**Using a real database.** If your data management needs are more complex than the usual single-file and single-spreadsheet scheme can handle, consider using a commercial **database** application for management of configurations, experimental data, and other important information. The advantages of a database are the abilities to index, search, and sort data with concurrent access from several locations but with explicitly regulated access to the data, thus enhancing security and reliability. The **Enterprise Connectivity Toolkit** (formerly the **SQL Toolkit**) from National Instruments enables LabVIEW to directly communicate with any **Open Database Connectivity (ODBC)**-compliant database application using **Structured Query Language (SQL)** commands. The toolkit is available for all versions of Windows as well as Macintosh and is compatible with nearly all major databases. It was originally created by Ellipsis Products and marketed as **DatabaseVIEW**.

The SQL Toolkit can directly access a database file on the local disk using SQL commands to read or write information, or the database can exist on a network—perhaps residing on a mainframe computer or workstation. You begin by establishing a *session*, or connection to



**Figure 14.5** The Enterprise Connectivity Toolkit connects LabVIEW to commercial databases. This example inserts simulated weather data into a climate database. (Example courtesy of Ellipsis Products, Inc.)

the target database(s), and then build SQL commands using LabVIEW string functions. The commands are then executed by the SQL Toolkit if you are connecting directly to a file or by a database application that serves as a *database engine*. Multiple SQL transactions may be active concurrently—an important feature, since it may take some time to obtain results when you are using a complex SQL command to access a very large database.

Figure 14.5 shows an example that simulates a year of weather in Boston, Massachusetts, and inserts each day's statistics into a dBASE database. The dataflow is pretty clear. First, a connection is made to the database by using the Connect VI for the data source *DBV DEMO DBASE*. The first two frames of the sequence (not shown) create the empty climate table in the database. Second, frame 2 begins by creating the year of simulated data. A dynamic SQL INSERT statement is also created by using wildcard characters (?) to designate the values that will be inserted later. In the For Loop, each field value is bound to its associated parameter in the wildcard list, and then the Execute Prepared SQL VI inserts data into the climate table. The dynamic SQL method can speed inserts up to 4 times over the standard convert, parse, and execute method. Third, the SQL statement reference is discarded in the End SQL VI. When the loop is complete, the last frame of the Sequence structure (not shown) disconnects from the database.

## Real-time analysis and display

LabVIEW's extensive library of built-in analysis functions makes it easy to process and display your newly acquired data in real time. You are limited by only two things: your system's performance and your imagination. Analysis and presentation are the things that make this whole business of virtual instrumentation useful. You can turn a voltmeter into a strip chart recorder, an oscilloscope into a spectrum analyzer, and a multifunction plug-in board into ... just about anything. Here, we'll look at the general problems and approaches to real-time analysis and display. In later chapters we will discuss particular applications.

Once again, we've got the old battle over the precise definition of **real time**. It is wholly dependent upon your application. If 1-min updates for analysis and display are enough, then *1 min* is real time. If you need millisecond updates, then *that's* real time, too. What matters is that you understand the fundamental limitations of your computer, I/O hardware, and LabVIEW with regard to performance and response time.

The kind of analysis you need to perform is determined by the nature of your signals and the information you want to extract (Table 14.1). Assuming that you purchase the full version of LabVIEW, there are about 400 analysis functions available. Other functions (and useful combinations of the regular ones) are available from the examples and from others who support LabVIEW through the Alliance Program. If you ever need an analysis function that seems obvious or generally useful,

**TABLE 14.1 Signal Types and Analysis Examples**

Signal type	Typical analysis
Analog—DC	Scaling Statistics Curve fitting
Analog—Time domain	Scaling Statistics Filtering Peak detection and counting Pulse parameters
Analog—Frequency domain	Filtering Windowing FFT/power spectrum Convolution/deconvolution Joint time frequency analysis
Digital on-off	Logic
Digital pulse train	Counting Statistics Time measurement Frequency measurement

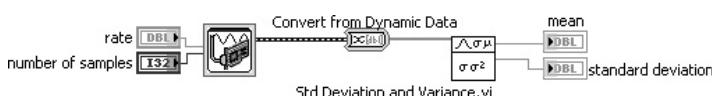
be sure to contact National Instruments (NI) to find out if it's already available. NI also takes suggestions—user input is really what makes this palette grow.

**Continuous versus single-shot data analysis.** Data acquisition may involve either **continuous data** or **single-shot data**. Continuous data generally arrives one sample at a time, like readings from a voltmeter. It is usually displayed on something such as a strip chart, and probably would be stored to disk as a time-dependent history. Single-shot data arrives as a big buffer or block of samples, like a waveform from an oscilloscope. It is usually displayed on a graph, and each shot would be stored as a complete unit, possibly in its own file. Analysis techniques for these two data types may have some significant differences.

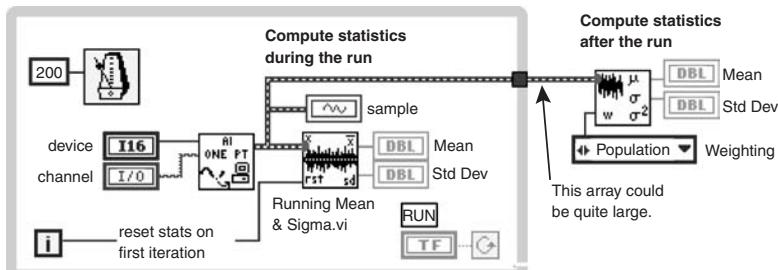
There is a special form of continuous data that we call **block-mode continuous data** where you continuously acquire measurements, but only load them into your LabVIEW program as a block or buffer when some quantity of measurements has accumulated. Multiple buffering or circular buffering can be carried out by any smart instrument, including DAQmx. The advantage of block-mode buffered operation is the reduced I/O overhead: You need to fetch data only when a buffer is half full, rather than fetch each individual sample. The disadvantage is the added latency between acquisition of the oldest data in the buffer and the transfer of that data to the program for processing. For analysis purposes, you may treat this data as either continuous or single-shot since it has some properties of both.

Here is an example of the difference between processing continuous and single-shot data. Say that your main interest lies in finding the mean and standard deviation of a time-variant analog signal. This is really easy to do, you notice, because LabVIEW just happens to have a statistical function called **Standard Deviation and Variance** which also computes the mean. So far, so good.

**Single-shot data.** A single buffer of data from the desired channel is acquired from a plug-in board. DAQmx returns a numeric array containing a sequence of samples, or waveform, taken at a specified sample rate. To compute the statistics, wire the waveform to the Standard Deviation and Variance function and display the results (Figure 14.6).



**Figure 14.6** Statistics are easy to calculate by using the built-in Standard Deviation and Variance function when data is acquired as a single shot (or buffer), in this case using the DAQmx Assistant.



**Figure 14.7** Gary had to write a special function, Running Mean & Sigma, that accumulated and calculated statistics during execution of a continuous data acquisition process. Building an array for postrun calculations consumes much memory.

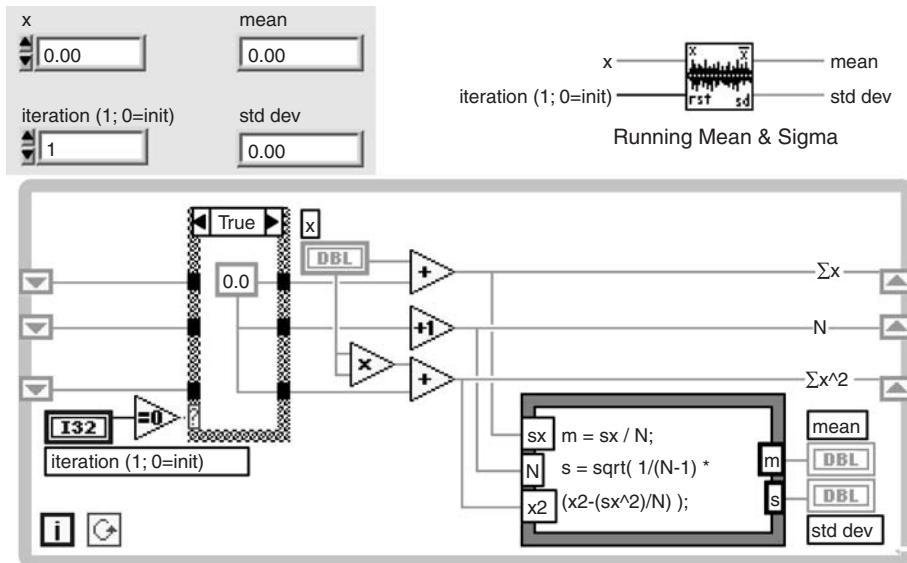
You might note that there is a conversion function before the input terminal to Standard Deviation. That's because DAQmx uses the dynamic data type, while most of the analysis library uses double-precision. The result is extra memory management and some loss of efficiency and speed, but it is currently unavoidable.

**Continuous data.** You can collect one sample per cycle of the While Loop by calling **AI Single Scan**, as shown in Figure 14.7. If you want to use the built-in Standard Deviation function, you have to put all the samples into an array and wait until the While Loop finishes running—not exactly a real-time computation. Or, you could build the array one sample at a time in a shift register and call the Standard Deviation function each time. That may seem OK, but the array grows without limit until the loop stops—a waste of memory at best, or you may cause LabVIEW to run out of memory altogether. The best solution is to create a different version of the mean and standard deviation algorithm, one that uses an **incremental** calculation.

Gary wrote a function called **Running Mean & Sigma** that recomputes the statistics each time it is called by maintaining intermediate computations in uninitialized shift registers (Figure 14.8). It is fast and efficient, storing just three numbers in the shift registers. A **Reset** switch sets the intermediate values to zero to clear the function's memory.

The idea came right out of the user's manual for his HP-45 calculator, proving that inspiration is wherever you find it. Algorithms for this and hundreds of other problems are available in many textbooks and in the popular *Numerical Recipes* series (Press 1990). You can use the concept shown here for other continuous data analysis problems.

National Instruments introduced another option for single-shot data analysis called the **Point-by-Point** library. It contains nearly all the analysis functions including signal generation, time and frequency domain, probability and statistics, filters, windows, array operations, and



**Figure 14.8** Running Mean & Sigma calculates statistics on an incremental basis by storing intermediate computations in uninitialized shift registers. The Reset switch clears the registers.

linear algebra. Internally, each function uses the same basic techniques we've just discussed for the Running Mean and Sigma VI. But there's some real numerical sophistication in there, too. Consider what it means to do an FFT calculation on a point-by-point basis. A simple solution would be to just save up a circular buffer of data and then call the regular FFT function on each iteration. Instead, NI implements the mathematical definition of the FFT, which is the integral of a complex exponential. This is, in fact, much faster, but makes sense only when performed on a sliding buffer. As a benchmark, you'll find that a 1024-point FFT runs about 50 percent faster in the point-by-point mode. It's an even larger difference when you're processing a buffer that's not a power of 2.

**Faster analysis and display.** Real-time analysis may involve significant amounts of mathematical computation. Digital signal processing (DSP) functions, such as the FFT and image processing functions, operate on large arrays of data and may require many seconds, even on the fastest computers. If execution time becomes a problem, you can:

- Make sure that you are using the most efficient computation techniques. Try to simplify mathematical expressions and processes and seek alternative algorithms.
- Avoid excessive array manipulation and duplication.

- Figure out ways to reduce the amount of data used in the calculations. Decimation is a possibility.
- Do the analysis postrun instead of in real time.
- Get a faster computer.
- Use a DSP coprocessor board.

Some of the options you may reject immediately, such as postprocessing, which is of little value when you are trying to do feedback control. On the other hand, if you really *need* that 8192-point power spectrum displayed at 5 kHz, then you had better be using something faster than the average PC. Always be sure that the analysis and display activities don't interfere with acquisition and storage of data.

**Reducing the volume of data.** Execution time for most algorithms is roughly proportional to the size of the data arrays. See if you can do something to reduce the size of your arrays, especially when they are to be processed by one of the slower functions. Here are some ideas:

*Sample at the minimum rate consistent with the Nyquist criteria and input filtering for your signals.* Many times, your data acquisition system will have to sample several channels that have widely different signal bandwidths. You may be able to rewrite the acquisition part of your program so that the low-bandwidth signals are sampled at a slower rate than the high-bandwidth signals. This may be more complex than using a single I/O function that reads all channels at once, but the reduction in array size may be worthwhile.

*Process only the meaningful part of the array.* Try to develop a technique to locate the interesting part of a long data record, and extract only that part by using the **Split Array** or **Array Subset** functions. Perhaps there is some timing information that points to the start of the important event. Or, you might be able to search for a critical level by using **Search 1D Array**, **Peak Detector**, or a **Histogram** function. These techniques are particularly useful for sparse data, such as that received from a seismometer. In seismology, 99.9 percent of the data is just a noisy baseline containing no useful information. But every so often an interesting event is detected, extracted, and subjected to extensive analysis. This implies a kind of triggering operation. DAQmx has a software triggering feature whereby data is transferred to the LabVIEW data space only if it passes some triggering criteria, including slope and level. This feature works for plug-in boards that don't have similar hardware triggering functionality.

*Data decimation is another possible technique.* **Decimation** is a process whereby the elements of an array are divided up into output arrays, much as a dealer distributes cards. The **Decimate 1D Array** function

can be sized to produce any number of output arrays. Or, you could write a program that averages every  $n$  incoming values into a smaller output array. Naturally, there is a performance price to pay with these techniques; they involve some amount of computation or memory management. Because the output array(s) is (are) not the same size as the input array, new memory buffers must be allocated, and that takes time. But the payoff comes when you finally pass a smaller data array to those very time-consuming analysis VIs.

**Improving display performance.** All types of data displays—especially graphs and images—tend to bog down your system. Consider using smaller graphs and images, fewer displayed data points, and less frequent updates when performance becomes a problem.

Figure 14.9 shows two ways of reducing the update rates of graphics, or anything else, be it an indicator or a computation. Figure 14.9A fixes the update rate of a graph in terms of a time interval by using the **Interval Timer VI**, described in Chapter 5.

Figure 14.9B permits a graph to update every  $n$  cycles of a data acquisition loop by testing the remainder output of the **Quotient and Remainder** function. Yet another way to reduce display overhead is to add an *update display* button connected to the Case structure in lieu of these automatic update techniques.

The displays could also appear in independent top-level VIs that receive data from the main acquisition loop via global variables. This is the client-server concept from Chapter 8, “Building an Application,” and it works well here. You write the data (probably an array) to a global variable in the data acquisition loop (the server). Then another VI containing graphs or other displays (the client) reads data from the global variable asynchronously. Analysis can be performed in either

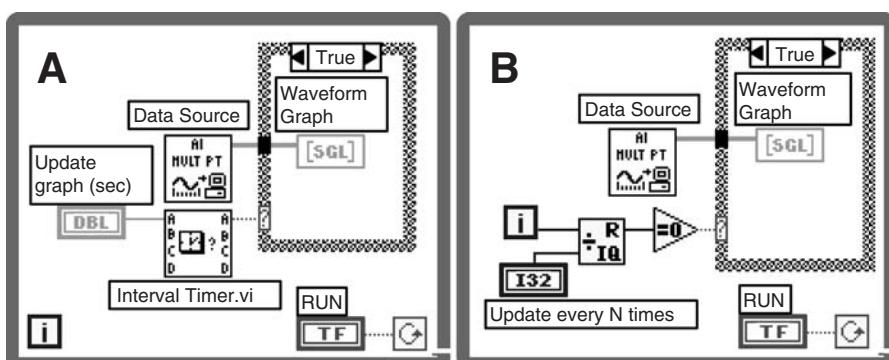


Figure 14.9 Limit the updating rates of graphics (or anything else) by using (A) the Interval Timer VI or (B) modulo arithmetic with the Quotient and Remainder function.

VI or both VIs. The advantage is that the client can be set to execute at lower priority than the server, as well as execute at a slower rate. Also, an arbitrary number of clients can run simultaneously. You could really get carried away and use another global that indicates that the display VI wants an update, thus creating a handshaking arrangement that avoids writing to the global on every acquisition. The disadvantage of the client-server scheme is that even more copies of the data are required, trading memory usage for speed.

**Graphics accelerators** are now common. All new PCs and Macs include graphics coprocessors that significantly reduce the overhead associated with updating the display (by as much as a factor of 50 in some cases). LabVIEW still has to figure out where the text, lines, and boxes need to go, and that takes some main CPU time. But the graphics board will do most of the low-level pixel manipulation, which is certainly an improvement. The LabVIEW Options item **Smooth Updates** makes a difference in display performance and appearance as well. Smooth updates are created through a technique called *off-screen bitmaps* where graphics are drawn to a separate memory buffer and then quickly copied to the graphics display memory. The intent is to enhance performance while removing some jumpiness in graphics, but smooth updates may actually cause the update time to *increase*, at least on some systems. Experiment with this option, and see for yourself.

## Sampling and Throughput

How much data do you need to acquire, analyze, display, and store in how much time? The answer to this question is a measure of system **throughput**. Every component of your data acquisition system—hardware and software—affects throughput. We've already looked at some analysis and display considerations. Next, we'll consider the input sampling requirements that determine the basic data generation rate.

Modern instrumentation can generate a veritable flood of data. There are digitizers that can sample at gigahertz rates, filling multi-megabyte buffers in a fraction of a second. Even the ubiquitous, low-cost, plug-in data acquisition boards can saturate your computer's bus and disk drives when given a chance. But is that flood of data really useful? Sometimes; it depends on the signals you are sampling.

### Signal bandwidth

As we saw in Chapter 13, "Sampling Signals," every signal has a minimum **bandwidth** and must be sampled at a rate at least 2 times this bandwidth, and preferably more, to avoid **aliasing**. Remember to include significant out-of-band signals in your determination of the sampling rate.

If you can't adequately filter out high-frequency components of the signal or interference, then you will have to sample faster. A higher sampling rate may have an impact on throughput because of the larger amount of raw data that is collected. Evaluate every input to your system and determine what sampling rate is really needed to guarantee high signal fidelity.

Sometimes, you find yourself faced with an overwhelming aggregate sampling rate, such as 50 channels at 180 kHz. Then it's time to start asking simple questions such as, Is all this data really useful or necessary? Quite often, there are channels that can be eliminated because of low priority or redundancy. Or, you may be able to significantly reduce the sampling rate for some channels by lowering the cutoff frequency of the analog low-pass filter in the signal conditioner. The fact that *some* channels need to go fast doesn't mean that they *all* do.

### Oversampling and digital filtering

Low-frequency analog signals give you some opportunities to further improve the quality of your acquired data. At first glance, that thermocouple signal with a sub-1-Hz bandwidth and little noise could be adequately sampled at 2 or 3 Hz. But by **oversampling**—sampling at a rate several times higher than the Nyquist frequency—you can enhance resolution and noise rejection. Noise is reduced in proportion to the square root of the number of samples that are averaged. For example, if you average 100 samples, the standard deviation of the average value will be reduced by a factor of 10 when compared to a single measurement. This topic is discussed in detail in “A Little Noise Can Be a Good Thing” in Chapter 13, “Sampling Signals.” We use oversampling for nearly all our low-speed DAQ applications because it’s easy to do, requires little extra execution time, and is very effective for noise reduction.

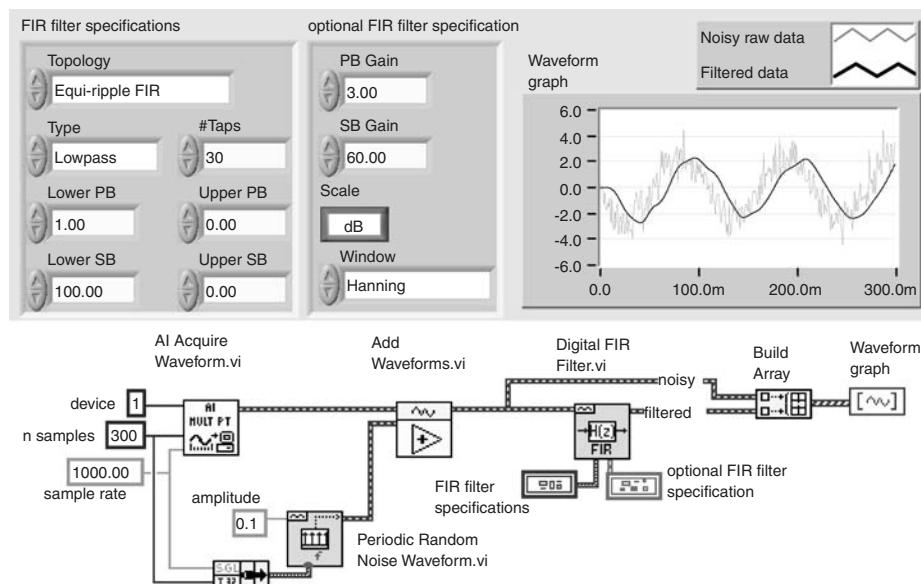
Once you have oversampled the incoming data, you can apply a digital low-pass filter to the raw data to remove high-frequency noise. There are a number of ways to do digital filtering in LabVIEW: by using the **Filter** functions in the analysis library or by writing something of your own. Digital filter design is beyond the scope of this book, but at least we can look at a few ordinary examples that might be useful in a data acquisition system. For more information, see *LabVIEW Signal Processing* (Chugani et al. 1998).

An excellent resource for filter design is National Instruments’ **Digital Filter Design Toolkit**, which is part of the Signal Processing Toolkit for LabVIEW and LabWindows/CVI. It is a stand-alone application for designing and testing all types of digital filters. You typically begin with a filter specification—requirements for the filter response in terms of attenuation versus frequency—and use the filter designer to compute the required coefficients that the LabVIEW VIs can use at

run time. The toolkit makes it easy by graphically displaying all results and allowing you to save specifications and coefficients in files for comparison and reuse. Displays include magnitude, phase, impulse, and step response, a  $z$  plane plot, and the  $z$  transform of the designed filter. Once you have designed a filter and saved its coefficients, an included LabVIEW VI can load those coefficients for use in real time.

**FIR filters.** If you are handling single-shot data, the built-in **Finite Impulse Response (FIR)** filter functions are ideal. The concept behind an FIR filter is a **convolution** (multiplication in the frequency domain) of a set of weighting coefficients with the incoming signal. In fact, if you look at the diagram of one of the FIR filters, you will usually see two VIs: one that calculates the filter coefficients based on your filter specifications, and the **Convolution** VI that does the actual computation. The response of an FIR filter depends only on the coefficients and the input signal, and as a result the output quickly dies out when the signal is removed. That's why they call it *finite* impulse response. FIR filters also require no initialization since there is no memory involved in the response.

For most filters, you just supply the sampling frequency (for calibration) and the desired filter characteristics, and the input array will be accurately filtered. You can also use high-pass, bandpass, and bandstop filters, in addition to the usual low-pass, if you know the bandwidth of interest. Figure 14.10 shows a DAQ-based single-shot application where



**Figure 14.10** A practical application of FIR low-pass filtering applied to an array of single-shot data, a noisy sine wave. Bandpass filtering could also be used in this case, since we know the exact frequency of interest.

a noisy sine wave is the signal. We applied a sinusoidal input signal, and then we added **Periodic Random Noise** to exercise our filter.

The **Digital FIR Filter VI** is set up to perform a low-pass filter operation. The **sampling frequency (fs)** for the filter VI is the same one used by the DAQ VI and is measured in hertz. This calibrates the **low cutoff frequency** control in hertz as well. Since the input signal was 10 Hz, we selected a cutoff of 50 Hz, which is sufficiently high to avoid throwing away any significant part of the signal. Topology is set for the FIR filter by specification, where you supply only the passband and stopband frequencies. Examine the graph and note that the filtered waveform is delayed with respect to the raw data. This is where the Digital Filter Design Toolkit comes in handy. You can choose the optimum cutoff frequency to yield the desired results. Filter design is always a compromise, so don't be disappointed if the final results are in some way less than perfect.

You can compensate for the delay in an FIR filter by routing the filtered signal through the **Array Subset** function and removing the required number of samples from the start of the waveform. That number is approximately equal to the number of taps; being the nonmathematical types, we usually test it out and see what the magic number is. Since the delay is invariant once you set the number of taps, FIR filters can effectively supply zero delay. (Technically, this is because they have *linear* phase distortion.) This is important when you need to maintain an absolute timing reference with respect to a trigger or another signal.

A **time window** generally should be applied to all signals prior to filtering or other frequency domain processing. Time windows reduce **spectral leakage** and other **finite-sequence length** artifacts in spectral estimation calculations such as FFTs and power spectra. Spectral leakage occurs because an FFT has discrete frequency bins. If the signal does not happen to precisely land in one of these bins, the FFT smears the signal energy into adjacent bands. The other problem, due to the finite data record length, shows up as extra energy spread out all over the spectrum. If you think about it, an arbitrary buffer of data probably does not start and end at zero. The instantaneous step from zero to the initial value of the data represents a transient, and transients have energy at all frequencies.

The job of the time window is to gradually force the start and end of your data to zero. It does so by multiplying the data array by some function, typically a cosine raised to some power, which by definition is zero at both ends. This cleans up much of the spectral leakage and finite-sequence length problems, or at least makes them more predictable. One side effect is a change in absolute amplitude in the results. If you use the **Scaled Window VI** from the Signal Processing >> Waveform

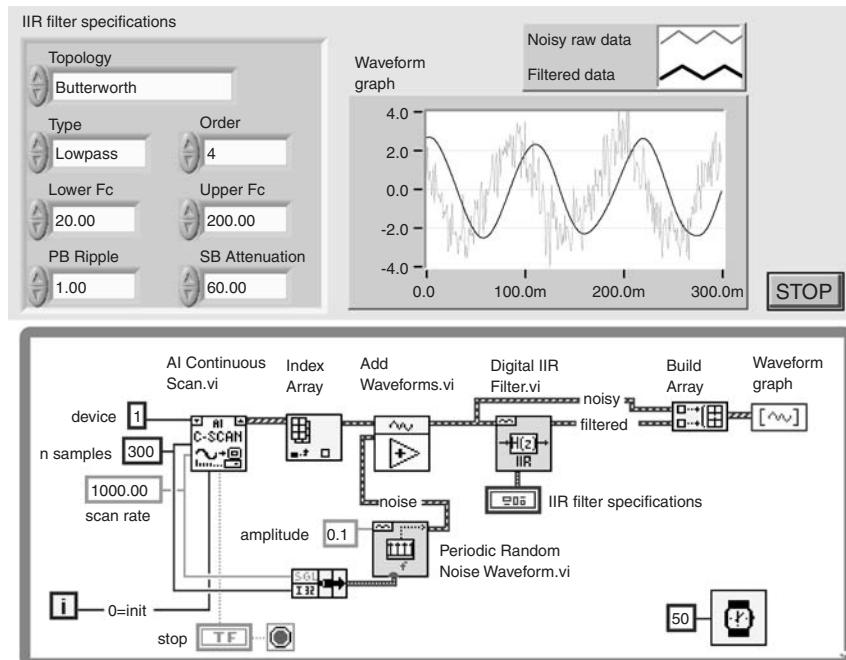
Conditioning palette, the gain for each type of window is properly compensated. The Digital FIR Filter VI is also properly compensated. There are quite a few window functions available, and each was designed to meet certain needs. You should definitely experiment with various time windows and observe the effects on actual data.

**IIR filters.** **Infinite Impulse Response (IIR)** filters are generally a closer approximation to their analog kin than are FIR filters. The output of an IIR filter depends not only on the *input* signal and a set of **forward coefficients**, but also on the *output* of the filter and an additional set of **reverse coefficients**—a kind of feedback. The resulting response takes (theoretically) an infinite amount of time to arrive at its final value—an asymptotic response just like the exponential behavior of analog filters. The advantage of an IIR filter, compared to an FIR filter, is that the feedback permits more rapid cutoff transitions with fewer coefficients and hence fewer computations. However, IIR filters have *nonlinear phase distortion*, making them unsuitable for some phase-sensitive applications. Also, you can choose a set of coefficients that make an IIR filter *unstable*, an unavoidable fact with any feedback system.

IIR filters are best suited to continuous data, but with attention to initialization they are also usable with single-shot data. In the Filter palette, you will find most of the classic analog filter responses in ready-to-use VIs: Butterworth, Chebyshev, Inverse Chebyshev, elliptical, and Bessel. If you are at all familiar with *RC* or active analog filters, you'll be right at home with these implementations. In addition, you can specify arbitrary responses not available using analog techniques. Of course, this gets a bit tricky, and you will probably want the Digital Filter Design Toolkit or some other mathematical tool to obtain valid coefficients.

Figure 14.11 shows the old NI DAQ function **AI Continuous Scan** VI acquiring seamless buffers of data from a single channel containing a 5-Hz sine wave. For interest, we added noise, just as in the previous FIR example. The **Digital IIR Filter** behaves in this application similarly to its analog counterpart, but we didn't need a soldering iron to build it.

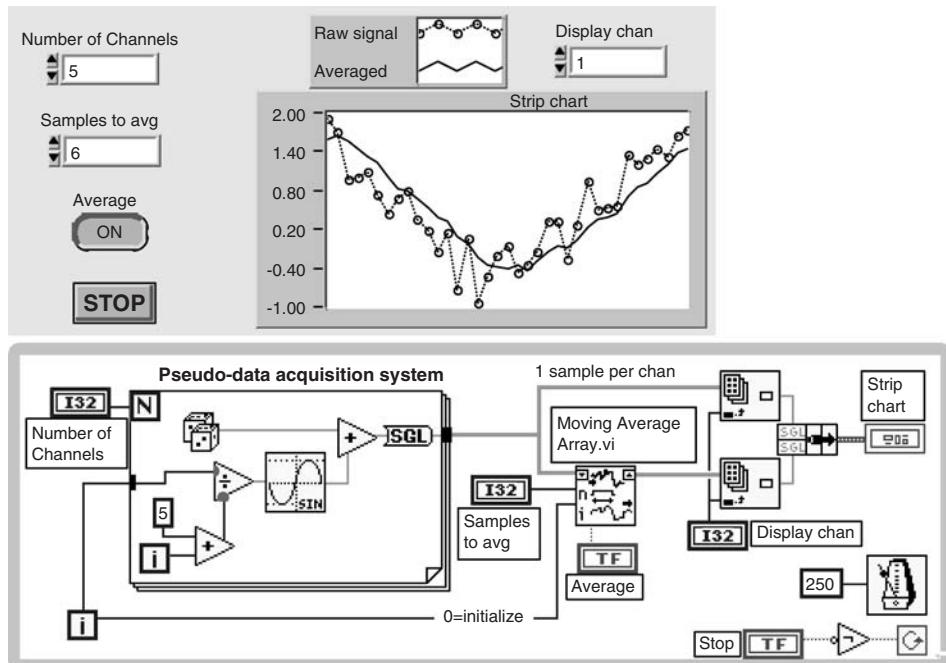
For classical filters such as this Butterworth filter, controls are quite similar to the design parameters for corresponding analog filters. In particular, you choose the **filter type** (high-pass, low-pass, etc.), and then you choose the **order**. Order, in the analog world, translates to the number of inductors and capacitors in the filter implementation. Higher order results in a sharper response, but as you might expect, the implementation requires additional computations. Sampling frequency and high- and low-cutoff frequencies are calibrated in hertz, as with the FIR filters.



**Figure 14.11** A continuous acquisition process with a Butterworth IIR filter. Once initialized, this digital filter works just as an analog filter does.

**Median filters.** If your data contains **outliers**—also known as *spikes* or *fliers*—consider the **Median Filter VI**. The median filter is based on a statistical, or nonlinear, algorithm. Its only tuning parameter is called **rank**, which determines the number of values in the incoming data that the filter acts upon at one time. For each location  $i$  in the incoming array, the filter sorts the values in the range  $(i - \text{rank})$  to  $(i + \text{rank})$  and then selects the median (middle) value, which then becomes the output value. The algorithm slides along through the entire data set in this manner. The beauty of the median filter is that it neatly removes outliers while adding no phase distortion. In contrast, regular IIR and FIR filters are nowhere near as effective at removing outliers, even with very high orders. The price you pay is speed: The median filter algorithm tends to be quite slow, especially for higher rank. To see how the median filter works, try out the Median Filtering example VI.

**Moving averagers.** **Moving Averagers** are another appropriate filter for continuous data. The one demonstrated in Figure 14.12, **Moving Avg Array**, operates on data from a typical multichannel data acquisition system. An array containing samples from each channel is the



**Figure 14.12** Demonstration of the Moving Avg Array function, which applies a low-pass filter to an array of independent channels. The For Loop creates several channels of data, like a data acquisition system. Each channel is filtered, and then the selected channel is displayed in both raw and filtered forms.

input, and the same array, but low-pass-filtered, is the output. Any number of samples can be included in the moving average, and the averaging can be turned on and off while running. This is a particular kind of FIR filter where all the coefficients are equal to 1; it is also known as a *boxcar* filter.

Other moving averagers are primarily for use with block-mode continuous data. They use local memory to maintain continuity between adjacent data buffers to faithfully process block-mode data as if it were a true, continuous stream.

After low-pass filtering, you can safely decimate data arrays to reduce the total amount of data. Decimation is in effect a **resampling** of the data at a lower frequency. Therefore, the resultant sampling rate must be at least twice the cutoff frequency of your digital low-pass filter, or else aliasing will occur. For instance, say that you have applied a 1-kHz low-pass filter to your data. To avoid aliasing, the time interval for each sample must be shorter than 0.5 ms. If the original data was sampled at 100 kHz (0.01 ms per sample), you could safely decimate it by a factor as large as  $0.5/0.01$ , or 50 to 1. Whatever you do, don't decimate

without knowledge of the power spectrum of your incoming signal. You can end up with exactly the same result as sampling too slowly in the first place.

### Timing techniques

Using software to control the sampling rate for a data acquisition system can be a bit tricky. Because you are running LabVIEW on a general-purpose computer with lots of graphics, plus all that operating system activity going on in the background, there is bound to be some uncertainty in the timing of events, just as we discussed with regard to timestamps. Somewhere between 1 and 1000 Hz, your system will become an unreliable interval timer. For slower applications, however, a While Loop with a **Wait Until Next ms Multiple** function inside works just fine for timing a data acquisition operation.

The *best* way to pace any sampling operation is with a hardware timer. Most plug-in boards, scanning voltmeters, digitizers, oscilloscopes, and many other instruments have sampling clocks with excellent stability. Use them whenever possible. Your data acquisition program will be simpler and your timing more robust.

Your worst timing nightmare occurs when you have to sample one channel at a time—and *fast*—from a “dumb” I/O system that has no ability to scan multiple channels, no local memory, and no sampling clock. Gary ran into this problem with some old CAMAC A/D modules. They are very fast, but very dumb. Without a smart controller built into the CAMAC crate, it is simply impossible to get decent throughput to a LabVIEW system. Remember that a single I/O call to the NI DAQ or NI GPIB driver typically takes 1 ms to execute, thus limiting loop cycle time to perhaps 1000 Hz. Other drivers, typically those based on peek and poke or other low-level operations including some DLL or CIN calls, can be much faster, on the order of microseconds per call. In that case, you at least have a fighting chance of making reliable millisecond loops.

If the aggregate sampling rate (total channels per second) is pressing your system’s reliable timing limit, be sure to do plenty of testing and/or try to back off on the rate. Otherwise, you may end up with unevenly sampled signals that can be difficult or impossible to analyze. It’s better to choose the right I/O system in the first place—one that solves the fast sampling problem for you.

### Configuration Management

You could write a data acquisition program with no configuration management features, but do so only after considering the consequences. The only thing that’s constant in a laboratory environment is *change*, so it’s silly to have to edit your LabVIEW diagram each time someone

wants to change a channel assignment or scale factor. Make your program more flexible by including configuration management VIs that accommodate such routine changes with a convenient user interface.

### What to configure

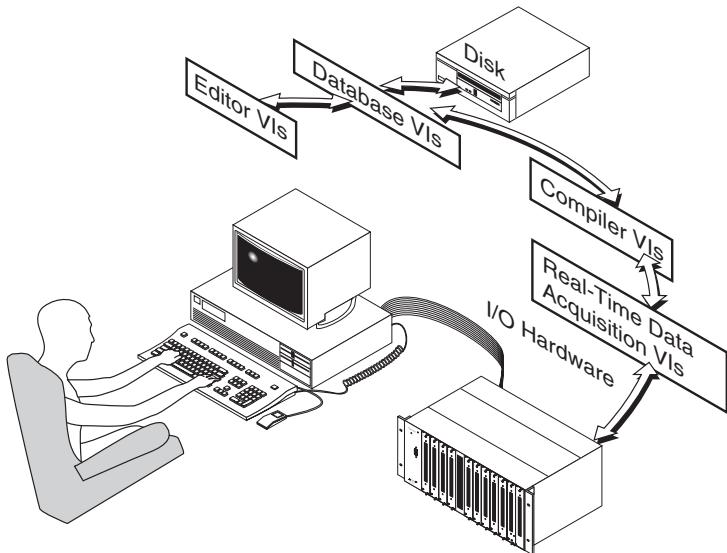
Even the simplest data acquisition systems have channel names that need to be associated with their respective physical I/O channels. As the I/O hardware becomes more complex, additional setup information is required. Also, information about the experiment itself may need to be inseparably tied to the acquired data. For instance, you certainly need to know some fundamental parameters such as channel names and sample intervals before you can possibly analyze the data. On the other hand, knowing the serial number of a transducer, while useful, is not mandatory for basic analysis. Table 14.2 is a list of configuration-related items you might want to consider for your system.

In a sense, all this information comprises a **configuration database**, and any technique that applies to a database could be applied here: inserting and deleting records, sorting, searching, and of course storing and fetching database images from disk. These are tasks for a configuration **editor**. Additionally, you need a kind of **compiler** or translator program that reads this user-supplied information, validates it, and then transmits it in suitable form to the I/O hardware and the acquisition program. (See Figure 14.13.)

The level of sophistication of such an editor or compiler is limited only by your skill as a programmer and your creative use of other applications

**TABLE 14.2 Items to Include in Configuration**

I/O hardware-related
Port number, such as GPIB board or serial port selection
Device address
Slot number, where multiple I/O modules or boards are used
Module or board type
Channel number
Channel name
Channel gain, scale factor, or linearization formula
Sampling rate; may be on a per-module or per-channel basis
Filtering specifications
Triggering parameters—slope, level, AC/DC coupling
Experiment-related
Experiment identifier; short ID numbers make good foundations for data file names
Purpose or description of the experiment
Operator's name
Start date and time
Data file path(s)
Transducer type or description for each channel
Transducer calibration information and serial number



**Figure 14.13** Process flow diagram for configuration management.

on your computer. The simplest editor is just a cluster array into which you type values. The most complex editor we've heard of uses a commercial database program that writes out a configuration file for LabVIEW to read and process. You can use the **SQL Toolkit** (available from National Instruments) to load the information into LabVIEW by issuing SQL commands to the database file. When the experiment is done, you might be able to pipe the data—or at least a summary of the results—back to the database. Do you feel adventurous?

Assuming that you're using DAQmx, much of this configuration information is addressed by the **Measurement and Automation Explorer (MAX)**. MAX is a convenient and feature-rich graphical-user interface that you use to maintain and test DAQmx tasks. From within LabVIEW, about all you need to do is to let the user pick from a list of predefined tasks. All the other information about each channel—scale factors, hardware assignments, and so forth—is already defined in the file, so your LabVIEW program doesn't need to keep track of it. Supplementary information, such as experimental descriptions, will of course be maintained by programs that you write.

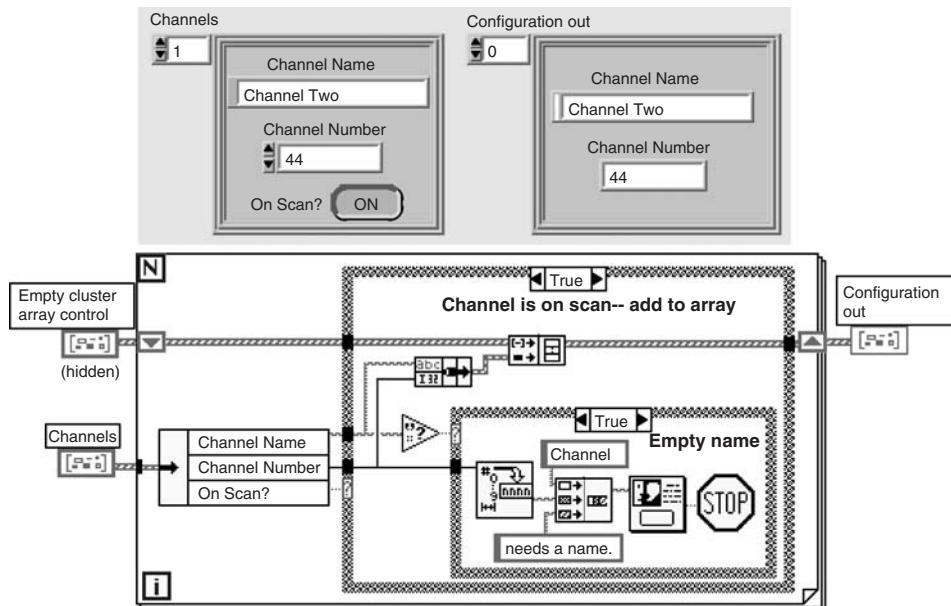
### Configuration editors

By all means, use MAX if you can. But it can't address every imaginable need, so you should know something about the alternatives. Aside from that rather elaborate application of a commercial database, there

are some reasonable ways for you to program LabVIEW as a configuration editor. There are two basic editor types: **interactive editors** and **static editors**. An *interactive editor* is a LabVIEW VI that runs while you enter configuration information, supplying you with immediate feedback as to the validity of your entries. A *static editor* is generally simpler, consisting of a VI that you run after entering information into all its control fields. If an error occurs, you receive a dialog box and try again. With either editor, once the entries are validated, the information is passed on to a configuration compiler (which may actually be part of the editor program). If the configuration is loaded from an external file, it may pass through an editor for user updates or directly to the compiler for immediate use.

**Static editors for starters.** Ordinary LabVIEW controls are just fine for entering configuration information, be it numeric, string, or boolean format. Since a typical data acquisition system has many channels, it makes sense to create a configuration entry device that is a **cluster array**. The cluster contains all the items that define a channel. Making an array of these clusters provides a compact way of defining an arbitrary number of channels (Figure 14.14).

One problem with this simple method is that it's a bit inconvenient to insert or delete items in an array control by using the data selection



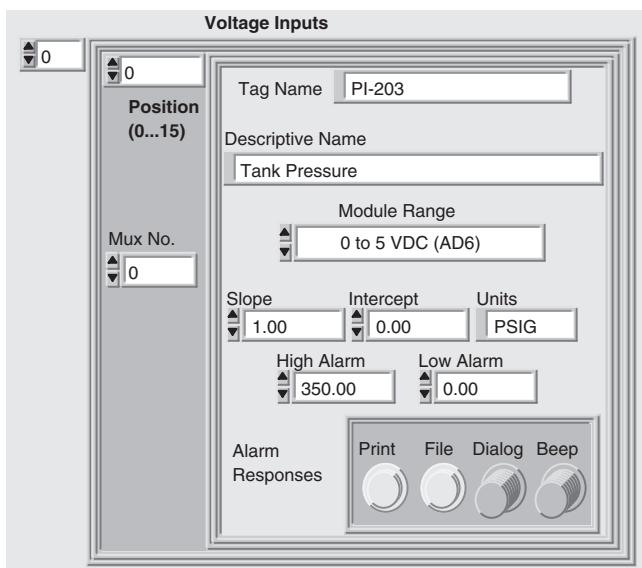
**Figure 14.14** A static configuration editor using a cluster array. It checks to see that a channel name has been entered and appends on scan channels to the output configuration array.

items in the array control pop-up. A solution is to include an **On Scan** switch, as I did in this example. Only when a channel is *on scan* is its configuration information passed along to the I/O driver. When a channel is *off scan*, the analog-to-digital converter (ADC) or digitizer may safely ignore that channel, thus saving some I/O operations. Make the switch turn red when a channel is off scan to ensure that the user doesn't accidentally leave a channel turned off.

The diagram for this editor checks to see that the channel name is not empty (you could also check for invalid characters, name too long, etc.). If the name is empty, a dialog pops up, telling the user, "Channel 17 needs a name." If the name is OK and the channel is on scan, then the name and channel number are appended to the output cluster array for use later on. You can also use the empty channel name as a switch to take the channel off scan.

Obviously, you can add as many items as you need to the channel description cluster, but the amount of error checking and cross-verification you could do will become extensive. Error checking is part of making your program robust. Even this simple example needs more checking: What if the user assigns the same channel number to two or more channels? What if two channels have the same name?

For more complex I/O systems, a nested structure of cluster arrays inside of cluster arrays may be required. Figure 14.15 configures



**Figure 14.15** Configuration for a more complicated I/O system (Opto-22 *Optomux*) requires nested cluster arrays with lots of controls.

Opto-22 analog input modules. The outer array selects a multiplexer (MUX) number. Each multiplexer has up to 16 channels that are configured by the inner cluster array. Since there were four possible alarm responses, we grouped them inside yet another cluster, way down inside. As you can see, the data structure becomes quite complex and might be unmanageable for a casual user. We've found that array controls are hopelessly confusing to some operators. This is a clear case of making the user interface convenient for the *programmer* rather than for the *user*. Such a structure might, however, be used as the output of a more user-friendly interactive editor.

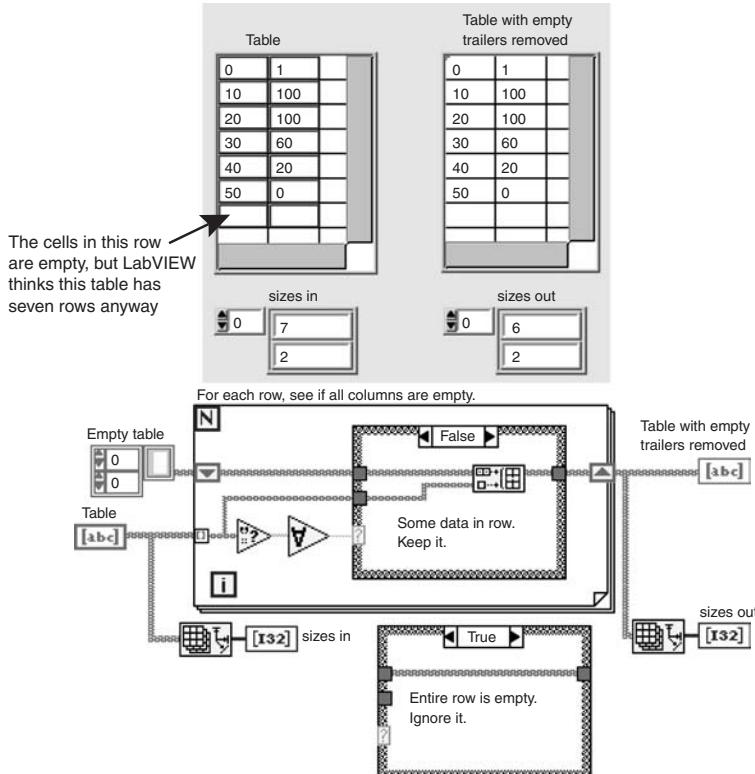
The LabVIEW **Table control** can also be used for configuration entries, and it's more likely that your users will understand it. As in a spreadsheet, columns can be assigned to each setup parameter with one row per channel. Tables have an advantage over cluster arrays because tables have high information density; many channel definitions will fit in a small screen area. But tables have one glaring disadvantage: Data validation is mandatory on each and every cell. Because a table is just a 2D string array, the user is free to type any characters into any cell. What happens when you are hoping for a numeric entry, but receive something nonnumeric?

Any time you write a loop to process a Table control, you will encounter the nefarious empty row problem. It occurs when the user types something into a cell and then deletes everything. Even though the cell is empty, the 2D array now has another element that contains an empty string. If you pass this 2D array to a loop for processing, it will attempt to process the empty strings. You can test for empty strings later or use the VI in Figure 14.16, **Remove Empty Table Rows**.

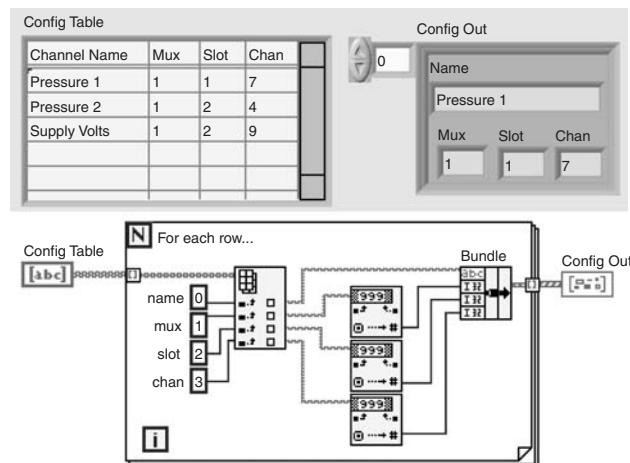
It removes any number of trailing empty rows. Size the Table control so that it displays only the proper number of columns; then hide the horizontal scroll bars to keep the user from entering too many columns of data.

Figure 14.17 is a static editor that interprets the data in a Table control. It does no error checking, but that could be added right after each of the Index Array functions. As a minimum, you have to verify that all numbers are in range and that the channel name is acceptable. Cross-checking for more complex data inconsistency gets really interesting. For instance, you might have to verify that the right number of channels is assigned according to the multiplexer type. Perhaps a future version of LabVIEW will include a more powerful Table control that has many of these data filter features, such as numeric validation.

**Interactive editors.** Interactive configuration editors are more versatile and more user-friendly than static editors because they provide instant feedback to the user. You can add pop-up windows, add



**Figure 14.16** This VI gets rid of empty table rows. Erasing a cell completely does not remove that row of the 2D array. But this VI does.

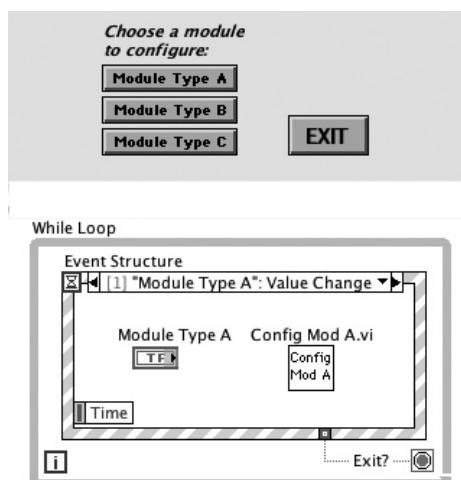


**Figure 14.17** This is a simple configuration editor that interprets the contents of a Table control and converts the data into a cluster array.

status displays, or even turn LabVIEW into a menu-driven system. Plan to do some extra programming if you want these extra features. Elaborate interactive editor projects are among the most challenging programming tasks we've ever tackled.

**Popup editors.** Say you have several different kinds of I/O modules, and they all have significantly different configuration needs. If you try to accommodate them all with one big cluster array, various controls would be invalid depending on which module was selected. You really need separate input panels for each module type if you want to keep the program simple.

One solution is to put several buttons on the panel of the main configuration VI that open customized configuration editor subVIs (Figure 14.18). Each button has its mechanical action set to **Latch When Released**, and each editor subVI is set to **Show front panel when called**. The editor subVIs do the real configuration work. Note that they don't really have to be any fancier than the static editors we already discussed. When you create one of these pop-up subVIs, remember to disable **Allow user to close window** in the Window Options of the VI Setup dialog. Otherwise, the user may accidentally close the window of the VI while it's running, and then the calling VI will not be able to continue.



**Figure 14.18** This configuration editor model uses subVIs that appear when an appropriate button is pressed. The subVI is the actual user interface (in this case, a configuration editor) and may do other processing as well. Each subVI is set to show the front panel when called.

Information may be returned by each editor subVI for further processing, or each editor can function as a stand-alone program, doing all the necessary I/O initialization, writing of global variables, and so forth. A worthy goal is for each editor to return a standard data structure (such as a cluster array), regardless of the type of I/O module that it supports. This may simplify the data acquisition and analysis processes (see the section entitled “Configuration Compilers” which follows).

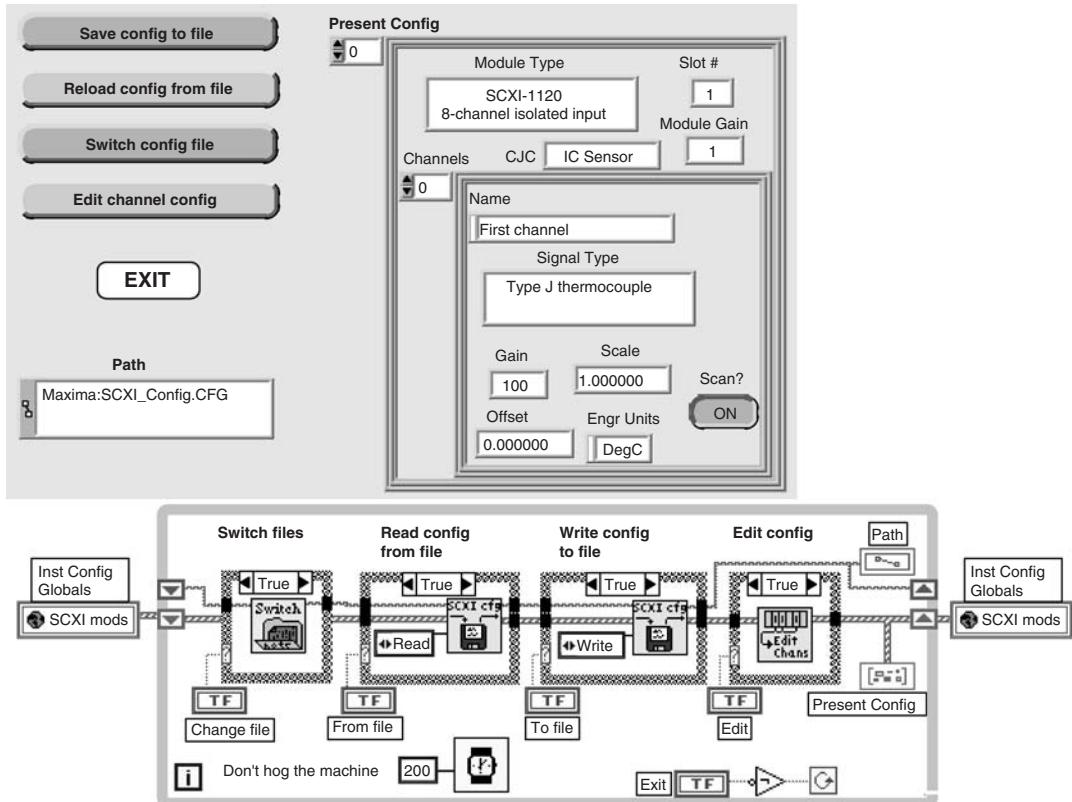
Another way you can accommodate modules with different configuration needs is to use **Property nodes** to selectively hide controls when they don’t apply. When the user selects module type *A*, the special controls for module *B* are hidden, and so forth. Note that you can selectively hide controls that are part of a cluster: Either pop up on the particular control in the cluster you wish to hide, and select **Create >> Property Node**, or get the array of control references to all the controls in the cluster and sort out which ones you want. There’s another way to tackle this problem: Place the clusters for each module on a tab control. This is the simplest and often the best solution.

Here is a model for an interactive editor hierarchy that Gary originally built to support SCXI analog input modules. It turned out to be quite versatile, and we’ve used it for many other kinds of instruments.

Figure 14.19 shows the window that appears when the user clicks a button on the top-level configuration manager VI. The figure also shows a simplified version of the diagram. This program relies on a global variable, Inst Config Globals, to pass the configuration cluster array along to other parts of the data acquisition system. The configuration data is loaded from the global at start-up and then circulates in a shift register until the user clicks the Exit button. Two subVIs act on the configuration data: **Read/Write SCXI Config** and **Edit SCXI Config**. The Read/Write SCXI Config VI can either load or store an image of the cluster array to a binary file. The file path is chosen by another subVI, **Change Config File**, which opens as a dialog box when called. It permits the user to pick an existing file or create a new one. The real work of editing the configuration occurs inside Edit SCXI Config.

Indeed, the tough part of this interactive editor is the Edit SCXI Config VI. Figure 14.20 shows the front panel, which was designed purely for user comfort and efficiency with no thought whatsoever for the complexity of the underlying program. So, where’s the diagram? It’s *way* too complicated to show here. It’s based on a state machine that looks at the present control settings and then modifies the display accordingly. For instance, if the user changes the **Module #** control, all the other controls are updated to reflect the present setting of channel 1 of that module.

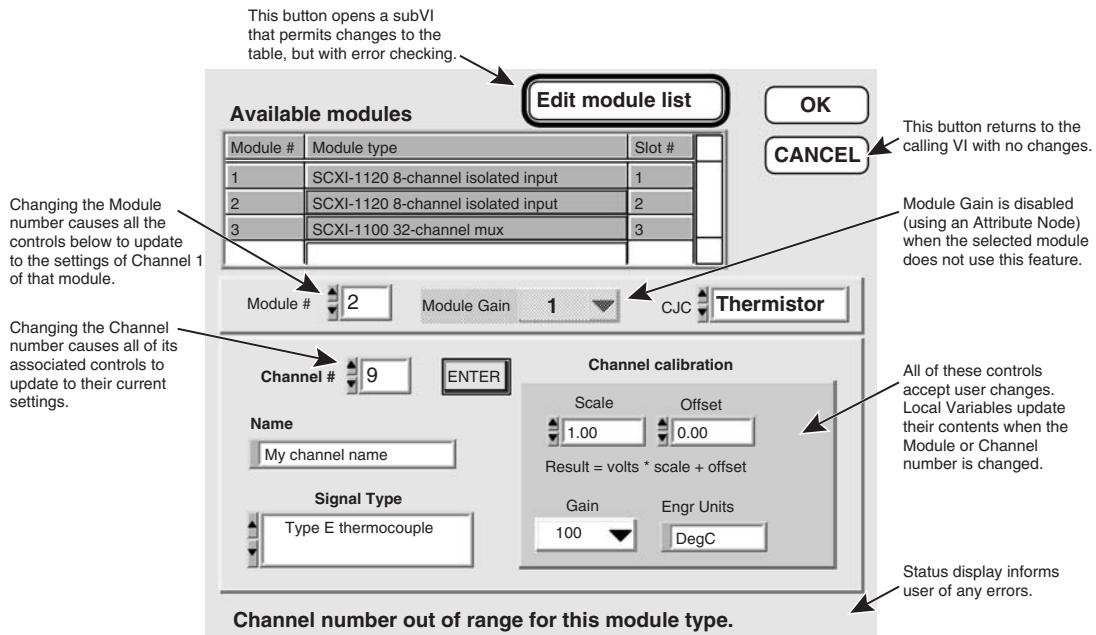
This is accomplished by reading the settings from the configuration cluster array and loading them into the controls with local variables.



**Figure 14.19** The SCXI Analog Config Dialog VI helps the user manage SCXI analog input modules. The configuration is stored in a global variable for use elsewhere in the data acquisition program, and it can be loaded from and saved to files. (False frames of all the Case structures simply pass each input to its respective output.)

Also, the **Module Gain** and channel **Gain** controls are selectively disabled (dimmed) by Attribute nodes. For instance, if the user picks an SCXI-1120 module, each channel has its own gain setting, so the **Gain** control is enabled while the **Module Gain** control is disabled.

This is the fanciest configuration editor that Gary could design in LabVIEW, and it uses just about every feature of the language. This editor is particularly nice because, unlike in the static editor with its cluster array, all the controls update instantly without any clumsy array navigation problems. However, the programming is very involved—as it would be in any language. So we suggest that you stick to the simpler static editors and use some of the ideas described here until you’re really confident about your programming skills. Then design a nice user interface, or start with my example VI, and have at it. If you design the

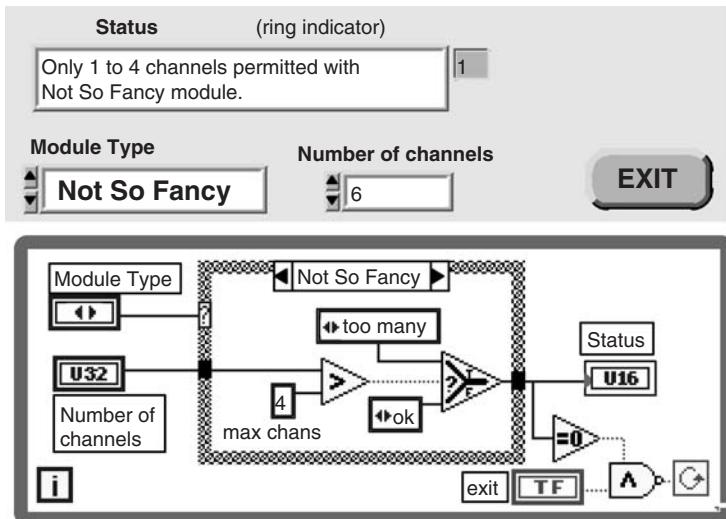


**Figure 14.20** Panel of the SCXI configuration editor, showing some of the features that you can implement if you spend lots of time programming.

overall configuration program in a modular fashion, you can replace the simple editor with the fancy editor when it's complete.

Only you can decide if great expenditures of effort are necessary. If you have only one or two users, or if they only have to deal with configuration management once a month, then it's probably not worth the effort. On the other hand, you may be able to develop an elaborate editor VI that can serve as a template for other purposes, thus leveraging your development efforts. When Gary wrote the SCXI configuration manager, he spent some time designing it with the intent that much of the code would be reused for other types of I/O hardware in his application. That's why the VI in Figure 14.19 has such a clean design.

**Status displays.** Since an interactive editor is a program that runs all the time, it can do such things as entry validation even as the user changes settings. You can give the user feedback by adding a status display that describes any errors or inconsistencies among the various settings. In Figure 14.21, for instance, you have a choice of several I/O modules, each having a certain limited range of channels. Any disagreement between module type and channel quantity needs to be rectified. A ring indicator contains predefined status messages. Item 0 is *OK* to



**Figure 14.21** How to use a ring indicator as a status display. The While Loop runs all the time, so the user's settings are always being evaluated. The ring indicator contains predefined messages—in this case, item 0 is *OK to continue*. Any real work for this configuration editor would be done inside the While Loop.

*continue*, while item 1 is the error message shown. The status has to be zero to permit the *Exit* button to terminate the While Loop.

You can also use a string indicator to display status messages, feeding it various strings contained in constants on the diagram. The string indicator can also display error messages returned from I/O operations or from an error handler VI. You would probably use local variables to write status messages from various places on your diagram, or keep the status message in a shift register if you base your solution on a state machine. That's what we did with the SCXI configuration editor. Some configuration situations require constant interaction with the I/O hardware to confirm the validity of the setup. For instance, if you want to configure a set of VXI modules, you might need to verify that the chosen module is installed. If the module is not found, it's nice to receive an informative message telling you about the problem right away. Such I/O checks would be placed inside the overall While Loop.

A good status message is intended to convey information, not admonish the user. You should report not only what is wrong, but also how to correct the problem. “ERROR IN SETUP” is definitely *not* helpful, although that is exactly what you get from many commercial software packages. **Dialogs** can also be used for status messages, but you should reserve them for really important events, such as confirming the overwriting of a file. It's annoying to have dialog boxes popping up all the time.

**Menu-driven systems.** When PCs only ran DOS, and all the world was in darkness, menu-driven systems were the standard. They really are the easiest user interfaces to write when you have minimal graphics support. The classic menu interface looks like this:

```
Choose a function:  
1: Initialize hardware  
2: Set up files  
3: Collect data  
Enter a number >_
```

In turn, the user's choice will generate yet another menu of selections. The good thing about these menu-driven prompting systems is that the user can be a total idiot and still run your system. On the other hand, an experienced user gets frustrated by the inability to navigate through the various submenus in an expedient manner. Also, it's hard to figure out where you are in the hierarchy of menus. Therefore, we introduce menus as a LabVIEW technique with some reluctance. It's up to you to decide when this concept is appropriate and how far to carry it.

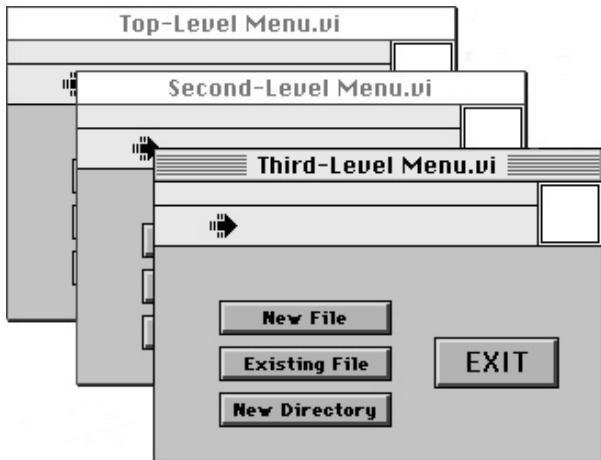
The keys to a successful menu-driven system are aids to navigation and the ability to back up a step (or bail out completely, returning to step 1) at any time. Using subVIs that open when called allows you to use any kind of controls, prompting, status, and data entry devices that might be required.

A LabVIEW menu could be made from buttons, ring controls, or sliders. If you use anything besides buttons, there would also have to be a *Do it* button. Programming would be much like the first pop-up editor example, earlier in this section. To make nested menus, each subVI that opens when called would in turn offer selections that would open yet another set of subVIs.

If you lay out the windows carefully, the hierarchy can be visible on the screen. The highest-level menu VI would be located toward the upper-left corner of the screen. Lower-level menus would appear offset a bit lower and to the right, as shown in Figure 14.22. This helps the user navigate through nested menus. LabVIEW remembers the exact size and location of a window when you save the VI. Don't forget that other people who use your VIs may have smaller screens.

### Configuration compilers

A **configuration compiler** translates the user's settings obtained from a configuration editor into data that is used to set up or access hardware. The compiler may also be responsible for storing and recalling old configuration records for reuse. The compiler program may or may not be an integral part of an editor VI.



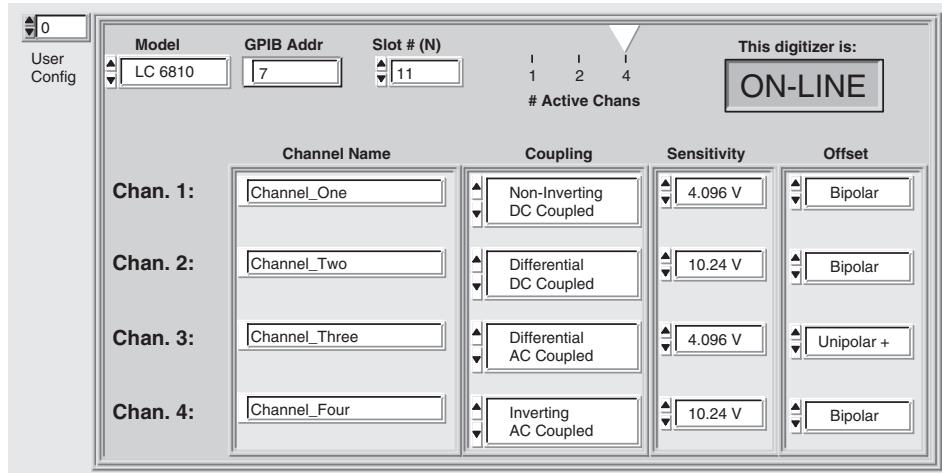
**Figure 14.22** Layout for hierarchical menu subVIs that helps users navigate. The Exit button returns you to the previous menu. You might use the VI Setup dialog to customize the window presentation; the Run arrow is not needed.

The control layout on a configuration editor's panel should be optimized for efficient user interaction. But those controls may not be very efficient when it comes time to send information to a real-time data acquisition driver or file storage VI. It would be very inefficient, for example, to have the driver interpret and sort arrays of strings that describe channel assignments every time the driver is called. Rather, you should write a compiler that interprets the strings at configuration time and creates a numeric array that lists the channel numbers to be scanned. Your objective is to write a program that does as little as possible during data acquisition and analysis phases, thus enhancing throughput.

Figures 14.23 through 14.25 illustrate a simple compiler that supports some digitizer modules. Each module has four channels with independent setups. Figure 14.23 is a cluster array into which the user has entered the desired setup information, perhaps by using an interactive editor VI. Channel setup information is in the form of four clusters, which makes it easy for the user to see all the setups at once. However, this is not an efficient way to package the information for use by an acquisition VI. Figure 14.24 shows a more desirable structure, where each channel is an element of an array. The compiler must make this translation.

Here are the steps to be taken by this configuration compiler, whose diagram appears in Figure 14.25:

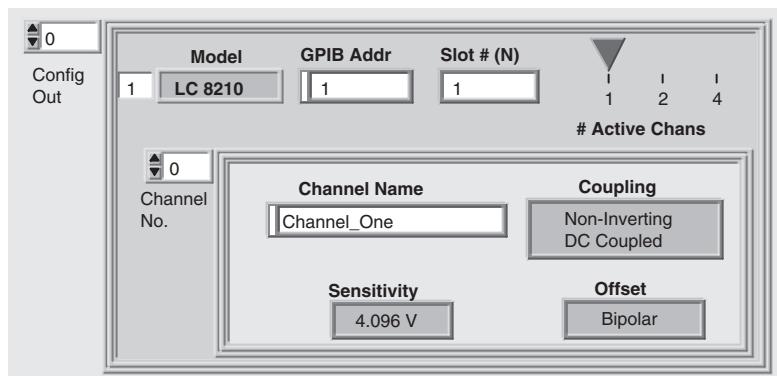
1. Convert the individual channel clusters into an array.
2. If the module is online, build the output configuration cluster and append it to the output array.



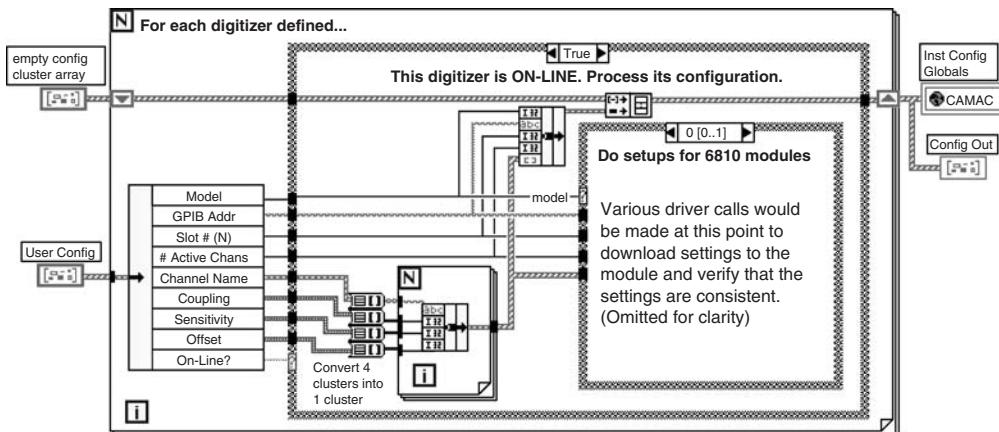
**Figure 14.23** The input to this configuration compiler for digitizers is a cluster array that a user has filled in. The four clusters (channel names, coupling, sensitivity, and offset) need to be checked for consistency and converted to arrays for use by the acquisition VIs.

3. Check the settings for consistency (for example, there may be limitations on sensitivity for certain coupling modes).
4. Initialize each module and download the settings.
5. Write the output configuration cluster array to a global variable which will be read by the acquisition VI.

We left out the gory details of how the settings are validated and downloaded since that involves particular knowledge of the modules and their driver VIs. Even so, much of the diagram is taken up by



**Figure 14.24** The compiler's output is a cluster array. Note that it carries all the same information as the input cluster array, but in a more compact form.



**Figure 14.25** Diagram for the simple compiler. Data types are converted by the inner For Loop. Settings are checked for consistency, and digitizers are initialized by the inner Case structure. If a digitizer is online, its configuration is added to the output array, which is passed to the acquisition VIs by a global variable.

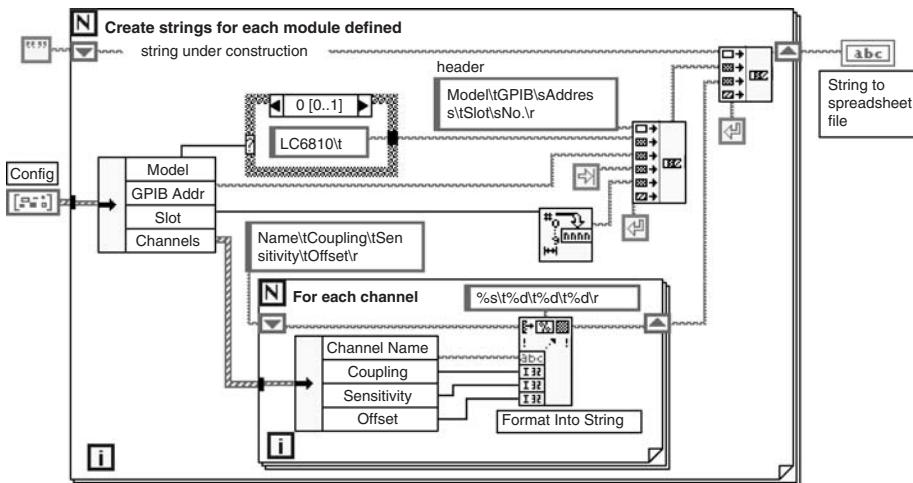
bundlers and unbundlers and other conversion functions that are required to reformat the incoming data. This is typical of these compilers, and it sometimes gets rather messy because of the number of items and special conditions that you have to deal with. Where possible, encapsulate related parts of the compiler in subVIs to make the diagram easier to understand and more compact. You would probably put all the driver functions for a given type of digitizer module in a subVI because they are logically related.

Somewhere in the edit or compile phase you should try to communicate with each I/O device to see if it responds, and report an error if it doesn't. It would be uncouth to permit the user to get all the way to the data acquisition phase before announcing that an important instrument is dead on arrival.

### Saving and recalling configurations

Another useful function that the compiler can perform is the creation of a permanent record of the configuration. The record might be a spreadsheet-format text file suitable for printing, a special format for use by a data analysis program, or a LabVIEW datalog or binary file that you can read back into the configuration program for later reuse. It makes sense to put this functionality in with the editors and compilers because all the necessary information is readily available there.

**Printable records (spreadsheet format).** Many systems we've worked on needed a hard-copy record of the experiment's configuration for the



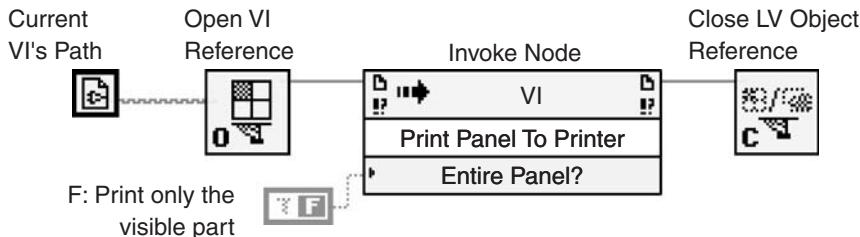
**Figure 14.26** This is a routine that converts the configuration cluster of Figure 14.24 into a tab-delimited text file for documentation purposes. As you can see, there's lots of string building to do.

lab notebook. The direct approach is to generate a text file that you can load and print with a spreadsheet or word processor. Add tab characters between fields and carriage returns where appropriate to clean up the layout. Titles are a big help, too. Figure 14.26 is a diagram that interprets the configuration cluster array of Figure 14.24 into tab-delimited text. There is a great deal of string building to do, so don't be surprised if the diagram gets a little ugly.

Here is what this text converter VI produces, as interpreted by a word processor with suitable tab stops. As you can see, most of the settings are written in numeric form. You could add Case structures for Coupling, Sensitivity, and Offset to decode the numbers into something more descriptive.

Model	GPIB Address	Slot No.	
LC6810	1	7	
Name	Coupling	Sensitivity	Offset
Channel One	7	2	0
Channel Two	1	4	2
Channel Three	0	3	1
Channel Four	4	5	1

Another way to print a configuration record is to lay out a suitable LabVIEW front panel and print that. You can use all the usual indicators as well as formatted strings displayed in string indicators, if that helps you get more information crammed onto one screen. The obvious



**Figure 14.27** The VI Server can do all sorts of run-time magic. Here, it's generating a printout of a panel—but only the part that's visible.

way to print the panel is to use the Print command in the File menu, but you can also print **programmatically**. Here's how. First, create a subVI with the displays you want to print. If there are extra inputs that you don't want to print, pop up on those items on the diagram and choose **Hide Control**. Second, turn on the **Print At Completion** option in the Operate menu to make the VI print automatically each time it finishes execution. All the print options in the VI Setup dialog will apply, as will the Page Setup settings. When you call this carefully formatted subVI, its panel need not be displayed in order to print.

Another way to generate a printout automatically is through the **VI Server**. This method is quite versatile since you can have the VI Server print any part of any VI, any time, any place. For instance, you can print the visible part of a VI's panel to the printer (Figure 14.27). Or you could route the front-panel image to an HTML or RTF file for archiving.

**Manual save and recall.** Your configuration manager should be able to recall previous setups for reuse. Otherwise, the user would have to type in all that information each time. You can use the **Datalogging** functions in the Operate menu (see "Writing Datalog Files" in Chapter 7). The Datalogging functions make it easy to retrieve setups at a later time. Each record has a timestamp and date stamp so you can see when the setup was created. Thus, your operator can manually save the current front panel setup and sift through old setups at any time.

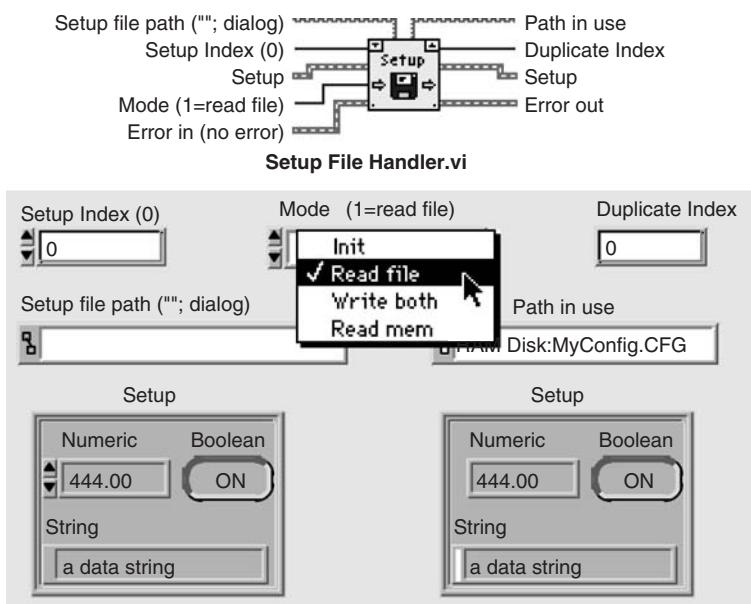
The crude way to save setups is by entering values into controls and then choosing **Make Current Values Default** from the Operate menu. However, this option is not available when the VI is running, so it's more of a system manager configuration item than something your operators would use. But these default values are appropriate for items that the operator doesn't need to change, and this is the easiest and most reliable way of saving setup data.

**Automatic save and recall.** We prefer to build the configuration editor with binary file support as previously described for the Read/Write

SCXI Config VI. This solution is very convenient for the user. Previous settings can be automatically loaded from a standard file each time the configuration editor is called, and then the file is updated when an editing session is complete.

Gary wrote the **Setup File Handler VI** to make this easy (Johnson 1995). It's also a handy way to manage front panel setups where you want to save user entries on various controls between LabVIEW sessions. For example, it's nice to have the last settings on run-time controls return to their previous states when you reload the VI. The Setup File Handler is an integrated subVI which stores a cluster containing all the setup data in a shift register for quick access in real time and in a binary file for nonvolatile storage. Multiple setups can be managed by this VI. For instance, you might have several identical instruments running simultaneously, each with a private collection of settings.

As you can see from the panel in Figure 14.28, the setup data is a cluster. Internally, setup clusters are maintained in an array. The element of the array to be accessed is determined by the Setup Index control. The **Setup** cluster is a typedef that you should edit with the Control Editor, and probably rename, for your application. You usually end up with a very large outer cluster containing several smaller clusters, one for each subVI that needs setup management.



**Figure 14.28** The Setup File Handler VI stores and retrieves setup clusters from disk. It's very useful for configuration management and for recalling run-time control settings at a later time.

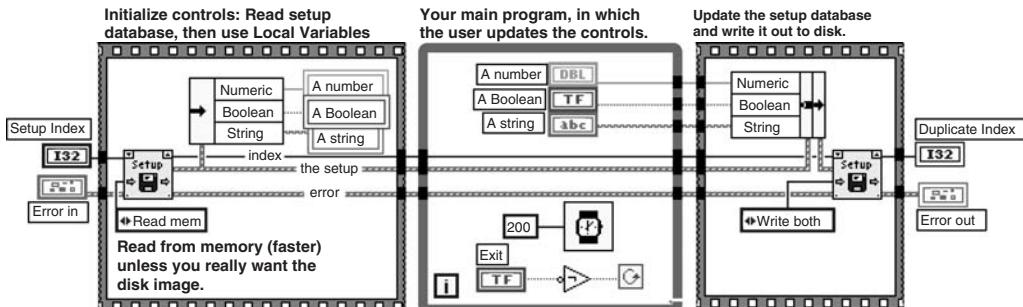
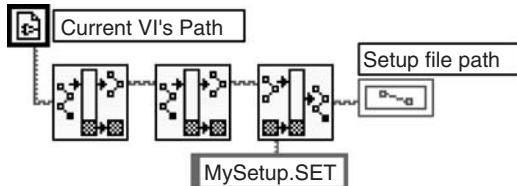


Figure 14.29 Using the Setup File Handler in a typical application.

To see how this all works, look at the **Setup File Handler** example in Figure 14.29. It begins by calling the Setup File Handler VI with Mode = Init to create or open a setup file. If the file doesn't exist, the settings in a default setup cluster are used for all setups. In most applications, all subsequent calls to Setup File Handler would be performed in subVIs. In the example subVI shown in Figure 14.29, the Setup File Handler is the source for control initialization data. Local variables initialize controls for the user interface, or the data might be passed to a configuration subVI. After the main part of the program finishes execution, the new control settings or configuration data are bundled back into the setup cluster, and the Setup File Handler updates the memory and disk images. Bundle and Unbundle by Name are very handy functions, as you can see. Try to keep the item names short in the setup cluster to avoid huge Bundlers.

Where should the setup file be saved? You can use the LabVIEW file constant **This VI's Path** to supply a guaranteed-valid location. This technique avoids a familiar configuration problem where you have a hand-entered master path name that's wrong every time you move the VI to a new computer. In your top-level VI, insert the little piece of code shown in Figure 14.30 to create a setup file path. If the current VI is in a LabVIEW library, call the **Strip Path** function twice as shown. If the VI is in a regular directory, call Strip Path once. This technique even works for compiled, stand-alone executables created by the LabVIEW Application Builder.

Another way to obtain a guaranteed-valid (and locatable) path is with the Get System Directory VIs written by Jeff Parker and published in *LabVIEW Technical Resource* (Parker 1994). He used CINs to locate standard directories, such as System: Preferences on the Macintosh or c:\windows on the PC. (You could simply assume that the drive and directory names are always the same, but sooner or later someone will rename them behind your back!)



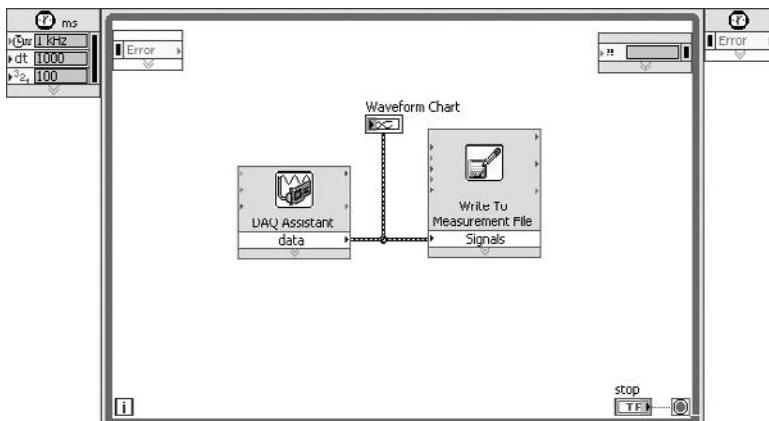
**Figure 14.30** Use the path manipulation functions Strip Path and Build Path to create a guaranteed-valid path for your setup files. If the current VI is in a LabVIEW library, call Strip Path twice, as shown. Otherwise, call it only once.

## A Low-Speed Data Acquisition Example

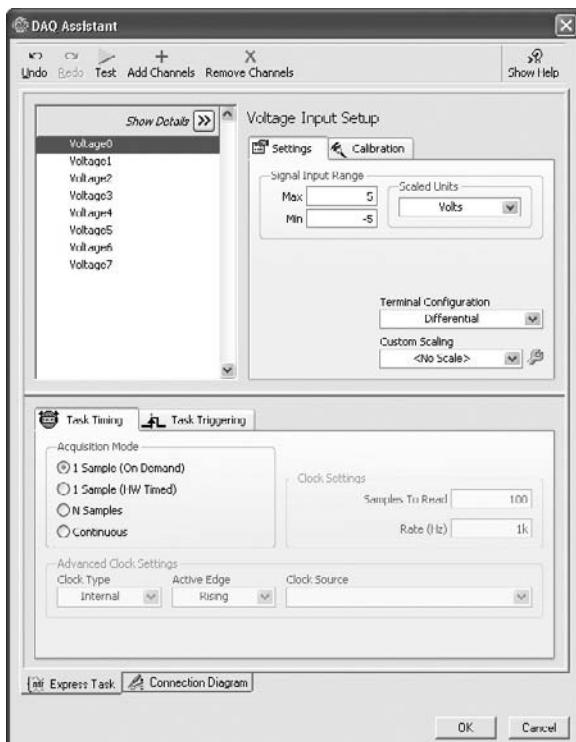
Life is unfair. You spend all this time reading about high-powered methods of building a data acquisition system, only to find out you need a system up and running *now*—not next week, not next month. What should you do? As we've stated before, your best bet is to program by plagiarizing. Sift through the examples that come with LabVIEW and see if there's something that might do at least 50 percent of the job. Or, search online and tap into the works of others. Eventually, you will build a personal library of VIs that you can call on to quickly assemble working applications. Following is an example that qualifies as a real solution to everyday data acquisition needs.

Here is a low-speed data acquisition VI that uses the **DAQ Assistant** and **Write to Measurement File** Express VIs. It's flexible enough to be the foundation of a somewhat larger application. It is intended for situations in which you need to acquire, display, and store data from a few analog channels at speeds up to perhaps 10 samples per second—a very common set of requirements. The diagram in Figure 14.31 shows the basic VI.

Express VIs provide a simple starting point when you have to get going fast. The DAQ Assistant walks you through configuring your DAQ measurement in just a few simple steps. Figure 14.32 shows the DAQ Assistant's configuration dialog configured to measure a single point from each of eight voltage channels. You can interactively test the configuration and change the input signal range if needed. The DAQ Assistant will even show you how to connect the signals for each channel. This example measures simple voltages, but the DAQ Assistant is fully capable of handling thermocouples, RTD's, strain gauges, and just about any measurement you might typically come across in the lab or the field. Once you're done, click OK and the DAQ Assistant will generate the LabVIEW code for your application. You can leave it as an Express VI or choose to Open Front Panel and convert the Express



**Figure 14.31** Diagram of a simple low-speed data acquisition VI using Express VIs. It's useful for a few channels at sampling rates up to about 10 Hz. This is a simple, but useful, example to which you can add features as required.

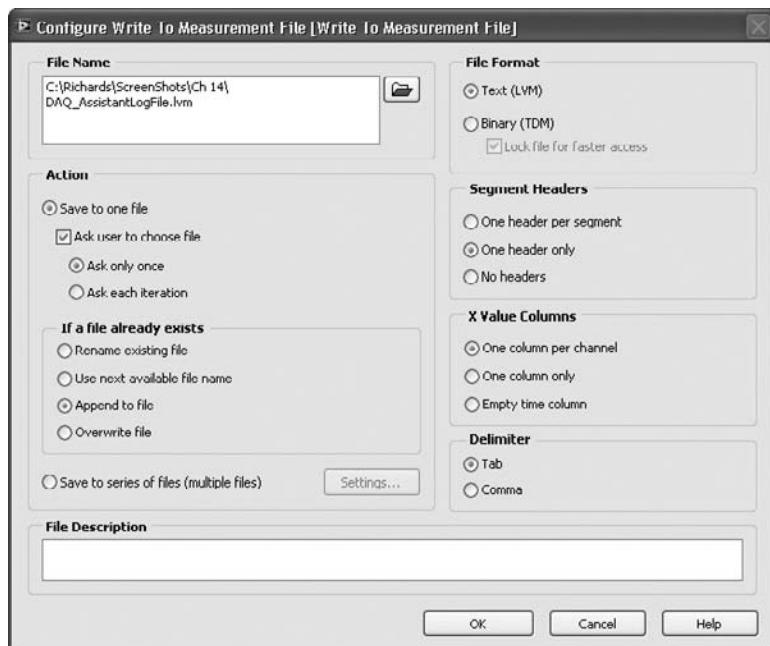


**Figure 14.32** The DAQ Assistant Configuration Wizard walks you through your measurement configuration. You can interactively test and modify until it's right.

block to a standard VI. Although once it's converted to a VI, you can no longer use the Wizard. In our example we've left it as an Express VI.

For timing we're using the Timed Loop set to run once every 1000 ms. We covered the Timed Loop in detail in Chapter 5, "Timing." The Timed Loop gives us a high-priority, fairly deterministic software timing mechanism for our data acquisition. Once the Timed Loop is started, it will interrupt any other process in order to run at its scheduled time.

We are writing the data to a text-based LabVIEW measurement file each iteration of the loop. The system overhead is fairly low, we only have eight channels at 1 Hz, and so we can afford to save them as text data. If we needed something faster or if file size were an issue, we could use the binary TDM format. Chapter 7, "Files," has some more information on these and other types of file I/O in LabVIEW. Figure 14.33 shows the configuration dialog for the Express VI **Write to Measurement File**. It creates a text-based LabVIEW measurement file with an informational header the first time it is called. Each subsequent time it appends the timestamped data to the file. This VI takes only a few minutes to put together, yet it can solve many problems. We hope you find it useful.



**Figure 14.33** Dialog for the Express VI Write to Measurement File. It creates a text-based LabVIEW measurement file (lvm) and appends timestamped data on each iteration.

## Medium-Speed Acquisition and Processing

Many applications demand that you sample one or more analog channels at moderate speeds, on the order of tens to thousands of samples per second, which we'll call *medium-speed acquisition*. At these speeds, you must have the I/O hardware do more of the time-critical work. The key to any moderate- and high-speed acquisition is **hardware-timed, circular buffered I/O using direct memory access (DMA)**. Although it sounds complicated, DAQmx sets this up for you transparently. Here's how the technology works:

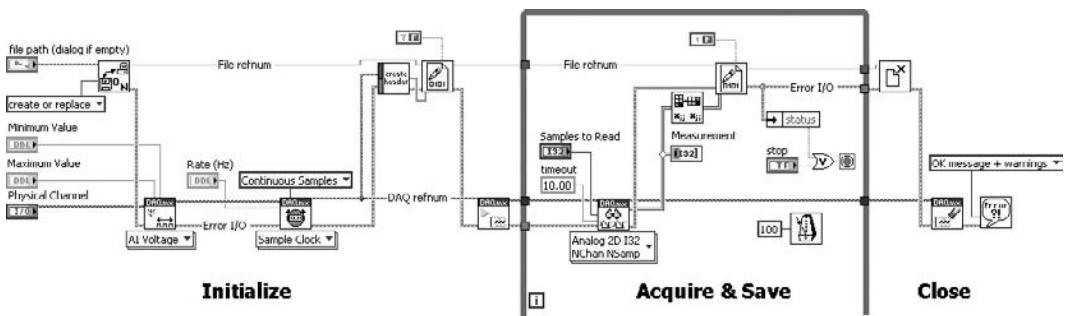
**Timed buffered I/O.** For DAQ applications running at speeds greater than a few tens of hertz, you must turn to hardware-timed buffered I/O. Every DAQ board has a piece of hardware called a **first-in, first-out (FIFO)** memory buffer that is located on the plug-in board. As data is read in, samples from the ADC are temporarily stored in the FIFO pending a bus transfer to your computer's main memory, giving the software extra time to take care of other business. An important piece of information is the size of the available FIFO buffer, which varies among models of plug-in boards. For instance, an E-series board has at least 512 words (a word is 16 bits, or one sample). If you are collecting data from a single channel at 10,000 samples per second, the FIFO buffer will fill up in 51.2 ms. If there is any overhead at all, the driver will not be able to get around to uploading the data before it is overwritten. If your LabVIEW program always had to generate or read each individual sample, there would be little hope of sampling at accurate rates much beyond 10 Hz. With buffered I/O, precision hardware timers control the real-time flow of data between memory buffers and the ADCs and DACs. Buffered operations can be triggered by hardware or software events to start the process, thus synchronizing the process with the physical world. LabVIEW transfers data between the plug-in board and a buffer whenever the buffer is full (for inputs) or empty (for outputs). These operations make maximal use of hardware: the onboard counter/timers, and any DMA hardware that your system may have.

**DMA.** The technique used to get memory off the board and in to memory is called direct memory access. DMA is highly desirable because it improves the rate of data transfer from the plug-in board to your computer's main memory. In fact, high-speed operations are often impossible without it. The alternative is interrupt-driven transfers, where the CPU has to laboriously move each and every word of data. Most PCs can handle several thousand interrupts per second, and no more. A DMA controller is built into the motherboard of every PC, and all but the lowest-cost plug-in boards include a DMA interface. The latest

boards have multiple DMA channels and *bus mastering* capability. When a board is allowed to take over as bus master (in place of the host CPU), throughput can increase even more because less work has to be done at interrupt time. With DMA, you can (theoretically, at least) acquire data and generate waveforms nearly as fast as your computer's bus can move the data.

**Circular buffered I/O.** Circular buffered I/O is a powerful technique because the buffer has an apparently unlimited size, and the input or output operation proceeds *in the background with no software overhead*. A circular buffer is an area of memory that is reused sequentially. Data is streamed from the DAQ hardware to the circular buffer by the DMA controller. When the end of the memory buffer is reached, the buffer wraps around and starts storing at the beginning again. Hopefully you've had time to read the data before it has been overwritten. The only burden on your program is that it must load or unload data from the buffer before it is overwritten. We really like circular buffered I/O because it makes the most effective use of available hardware and allows the software to process data in parallel with the acquisition activity.

Figure 14.34 shows the DAQmx example VI **Cont Acq&Graph Voltage To File (Binary).vi** (examples >> DAQmx >> Analog In >> Measure Voltage.llb). This VI follows our canonical DAQ model where we initialize outside the loop, acquire and process (or save) inside the loop, and close out our references after the loop. It's all tied nicely together by chaining the error I/O to enforce dataflow. When we configure our sample clock, DAQmx configures the board for hardware-timed I/O. By enabling continuous samples, DAQmx automatically sets up a circular buffer. Data is streamed to disk inside the loop as raw binary data. This example can easily stream data from multiple channels at 10,000 samples per second until your hard drive fills up. For more DAQ



**Figure 14.34** The Cont Acq&Graph Voltage To File (Binary).vi. This medium-speed example can acquire and save binary data to disk at 10,000 samples per second until you run out of storage.

examples you can modify to fit your application needs, look at the DAQmx examples shipping with LabVIEW.

## Bibliography

- Chugani, Mahesh, et al.: *LabVIEW Signal Processing*, Prentice-Hall, Englewood Cliffs, N.J., 1998.
- Johnson, Gary W.: "LabVIEW Datalog Files," *LabVIEW Technical Resource*, Vol. 2, No. 3, Summer 1994. (Back issues are available from LTR Publishing.)
- Johnson, Gary W.: "Managing Front-Panel Setup Data," *LabVIEW Technical Resource*, Vol. 3, No. 1, Winter 1995. (Back issues are available from LTR Publishing.)
- Parker, Jeff: "Put config Files in Their Place!" *LabVIEW Technical Resource*, Vol. 2, No. 3, Summer 1994. (Back issues are available from LTR Publishing.)
- Press, William H., et al.: *Numerical Recipes in C*, Cambridge Press, New York, 1990.

*This page intentionally left blank*

## LabVIEW RT

In just a few years, **LabVIEW RT** has grown from a niche product to become a staple of automation. LabVIEW RT on one of National Instruments' many industrial computers conveniently fills the gap between a **Programmable Logic Controller (PLC)** and a PC. The result is a whole new class of industrial controller called the **Programmable Automation Controller (PAC)**. PACs fill a need for high-loop-rate, closed-loop control systems requiring advanced analysis and integration with the corporate network. And PACs do it for a reasonable cost.

Whether you're programming a PAC or just need a reliable DAQ system in the lab, LabVIEW RT has the hardware and software to fill most needs. Applications written in LabVIEW RT are deployed on an embedded target (one with no keyboard, mouse, or display) and run under a commercial **real-time operating system (RTOS)**. You build your application under Windows on a regular PC, and then you download the code via Ethernet to run on the real-time target system. All the functionality of regular LabVIEW is available. LabVIEW RT and PACs let us attack a whole new class of applications while maintaining the familiar ease of development and use for which LabVIEW is famous.

In this chapter, we'll start with some background information about real-time systems and LabVIEW RT, and then we'll cover the important issues that crop up when you develop real-time applications, including timing, multithreading, task scheduling, and task priorities. Let's start off by defining what we mean by real time.

### Real Time Does Not Mean Real Fast

There are several characteristics that define a real-time application. Too often we hear the term *real time* misused as meaning *real fast*. LabVIEW G code will not run any faster under a real-time operating

system than it will under any other operating system! LabVIEW is compiled into machine code and runs at the speed of the processor. Only a faster CPU will give you faster code. What the RTOS gives you that your desktop system can't is **determinism**, which is perhaps the most important factor in defining real time.

Determinism means that you can accurately predict or determine when a section of your program will execute. If you are operating a closed-loop controller on the factory floor, you may have a calculation and a set point update that you need to make after each analog input reading. It's just as important to your process that the update be made within a critical time window (meet the deadline) as it is that your program make the correct calculation. How late or early the code is in actually running with respect to the desired time is known as its **latency** or worst-case response time. If your process will fail if you're 1 s late, then 1 s is your real-time latency tolerance. As a benchmark, with LabVIEW RT and software timing alone we can easily achieve 1-kHz closed-loop control with latencies of less than 50  $\mu$ s. With hardware timing, latency can drop to nanoseconds.

To look at this determinism issue in another way, let's consider the requirements of an embedded controller for a robot on a factory assembly line. As product comes down the line, the robot needs to put its arm in the right place at exactly the right time to do its job. The correctness of the robot's program depends as much on the robot being on time as it does on making the right calculation to be in place.

A real-time system can be classified as **hard real time** or **soft real time** based partly on the consequences of missing a timing deadline. A soft real-time system allows for some deadlines to be missed with only a slight degradation in performance but not a complete failure. In a soft real-time system, the assembly line may be designed so that the product will stay in position until the robot can come do its job. The assembly line may have to slow down to compensate for any delays that result from the robot being a little late, but the end result is that no harm is done other than a little less product comes off the line.

In a hard real-time system, the robot would have only a very small time window when the product was there. If the robot did not do its job at the right time, the product would be gone or worse. When a system has tasks that have to be run on time or else the system will fail, it is a hard real-time system. In some cases, this failure to meet a deadline could result in the loss of life and/or property.

An RTOS is an operating system (OS) that has been designed to do one thing: run your code in a predictable manner with minimal latency. That means it meets the requirements for *hard* real time. In contrast, all modern desktop operating systems operate on a fairness basis. Each application or process gets a little bit of time on the CPU regardless of

its priority. LabVIEW on a desktop computer is subject to the timing uncertainties caused by the desktop OS, and we cannot be sure that the OS will not suspend the execution of our most important, time-critical VI while it goes out to perform some routine maintenance. We have all seen this happen when a background event briefly interrupts a high-priority acquisition and control loop. Clearly, this kind of *soft* real time cannot be used to generate or respond to time-critical tasks in which the latency needs to be better than perhaps a few hundred milliseconds.\*

Because LabVIEW RT runs on an RTOS, we can write software that will meet a guaranteed deadline. But this predictability comes with a price: We have to pay close attention to threading, priorities, memory management, and other tricky technical issues that we'll cover later in this chapter. Most important, we need to have realistic expectations.

Designing software for a real-time system requires different goals from designing software for use with a desktop OS. Rapid prototyping is one of the areas in which LabVIEW excels, but it can lead to some bad habits. Too often we get in the habit of just throwing things together until we get the right answer, but without paying much attention to how efficiently our code executes. Because performance is just as important in a real-time system as having the right answer, we need to take a different approach to software design.

Sometimes you just need a headless (no GUI) version of an existing LabVIEW program. In that case, you can simply use LabVIEW RT to recompile your existing code onto an RT-compatible target, and you are done. LabVIEW RT is fully cross-platform-compatible with all other versions of LabVIEW and makes building headless embedded systems easy. On the other hand, if you are interested in building control systems that can meet time-critical demands, then read on. We'll try to show you the basic information that you'll need to build a system to meet your real-time demands.

## RT Hardware

The LabVIEW RT development environment runs on a Windows host. The host serves as the local storage for the RT applications during development as well as provides the user interface. When the Run button is pressed, the LabVIEW VIs are downloaded to the embedded target, where they are compiled and run on their own dedicated processor

---

\*There's nothing wrong with using a desktop system in a soft real-time application; there are perhaps millions of such systems in operation today. You only have to prove that the system meets your requirements for determinism under all circumstances. Dedicated a computer to a control task, deleting unneeded applications, and keeping users away leave it lightly loaded, thus reducing latency.



**Figure 15.1** The first platform for LabVIEW RT was the 7030-series RT DAQ card with an onboard 486 processor and analog and digital I/O. Today the PCI 7041/6040E uses a 700-MHz PIII with 32-Mbyte RAM and 32-Mbyte Flash. (*Photo courtesy of National Instruments.*)

under a real-time OS. The user interface runs only on the host, not on the embedded target. Normal LabVIEW user-interface interaction takes place over a TCP/IP (Ethernet) communications link using **Front Panel Protocol (FPP)** routines that National Instruments provides in LabVIEW RT.

LabVIEW RT runs on the **PCI 7041/6040E** series RT DAQ boards, on selected PXI crate controllers, Compact Fieldpoint controllers, Compact Vision systems, and even off-the-shelf dedicated desktop PC hardware (Figures 15.1 and 15.2). The 7041 intelligent DAQ board plugs



**Figure 15.2** Programmable automation controllers (PACs) combine PLC ruggedness with PC-like functionality. From left to right: CompactRIO, Compact Vision, PXI, Compact FieldPoint. (*Photo courtesy of National Instruments.*)

into an existing computer's PCI slot. Each intelligent RT DAQ card consists of a main processor card and a daughter card for data acquisition tasks. The processor card has a PIII/700-MHz CPU with related support circuitry and 32 Mbytes of RAM for user programs, and 32 Mbytes of permanent storage. The daughter card is National Instruments' standard 6040E 12-bit MIO card that has been adapted to fit onto the processor card. It's a handy way to upgrade an existing system to enhance time-critical performance. It's also a way to add parallel processing power to a single PC, since you can run multiple processor cards simultaneously.

LabVIEW RT on a rugged CompactRIO, Compact Vision, or Compact FieldPoint provides an industrial-strength hardware platform with a reliable OS and flexible programming environment. LabVIEW RT on one of National Instruments' **PXI** controllers offers the greatest speed and flexibility. All the PXI DAQ modules that work with DAQmx and most modular instruments will work in real time under LabVIEW RT with the PXI controller. If you already have a PXI system, you should be able to upgrade it to real-time performance without a major rework of any hardware. You can also use a **desktop PC** as an RT controller and use your existing PCI DAQ cards. Any Pentium 4-based computer system with the same PCI and Ethernet chip set as the PXI board *should* be capable of running LabVIEW RT applications. Check out NI's website ([www.ni.com](http://www.ni.com)) for complete and up-to-date information.

The sole purpose of a real-time system is to perform time-critical or real-time tasks. In LabVIEW RT the user interface has been stripped out to allow all the processor cycles to be used for the real-time task. All user interaction with the RT card and communication between the host and the plug-in RT card are done through shared memory on the 7041 and over Ethernet on the modular platforms. We can use this shared memory for high-speed communications when necessary, but for most of our routine tasks we can just use LabVIEW's built-in TCP/IP functions and the high-level VI Server functions.

The only way that you can interact with the RT system on the PXI crate is over Ethernet. LabVIEW RT has *all* of LabVIEW's communication routines, including the VI Server, TCP/IP, and Web server. Just make sure that your communication routines run in a separate loop at a lower priority than your real-time tasks. You don't want your user interface to get in the way of your time-critical tasks. That's why we moved to a real-time system in the first place.

Since the RT card does not have an Ethernet port, the LabVIEW team has implemented TCP/IP and VI Server functionality through shared memory. This is just one of the many ways LabVIEW takes care of the details for us so that we can concentrate on our application and don't worry about the implementation. Some of the VI Server

functions, such as Make Current Values Default, don't make any sense on a card without any local storage, and so they have been implemented only on the PXI system. These minor details are covered in the LabVIEW RT manuals.

One of the first things you will notice after you have booted up your PXI crate into the real-time environment is that the graphical display is gone. You are left with a black, DOS-like screen and a cursor that doesn't respond to keyboard input. In the real-time system, all user-interface code is stripped out, and only the compiled block diagram code is left. On the PXI system there is a *printf* function that will print a line to the monitor for debugging during development, but other than that there is no resident user-interface at all! All user-interface activity is handled by the Windows-based development system over the Ethernet link.

## Designing Software to Meet Real-Time Requirements

Ugly code is ugly code! Don't download some poorly designed spaghetti code onto a LabVIEW RT system and expect to have it magically work better. Chances are it will be worse. Programming an application that will meet a hard real-time deadline requires you to write code that will execute efficiently. Fortunately, programming in LabVIEW RT is not any different from programming in regular LabVIEW, and we have the entire suite of LabVIEW tools to work with. All you have to do is to learn some new guidelines, many of which will probably improve the performance of your non-RT applications as well.

A good model when you are designing software for deterministic behavior is to look at the techniques used in hardware design. In a very real sense, what we are trying to do with our software is to emulate a hardware task. Many real-time tasks started out as hardware tasks but needed the flexibility that a software design can bring. The art is to implement the design in software without paying too much of a performance penalty.

When you need to design a circuit, you need to know the inputs and outputs just as in software. You can predict how each device you put into your circuit will behave because you know the physical characteristics of each device and you can verify the performance with hardware probes such as oscilloscopes and logic analyzers.

This same modularization and performance verification is exactly how you must go about designing your software for a real-time system. Since time is a critical element of a real-time system specification, you need to know exactly how much time each subVI requires to execute.

When you design a real-time system, one of the obvious questions you need to ask yourself is, What is our *real-time* task? Suppose we have a process controller that needs to run at 1 kHz and we also have

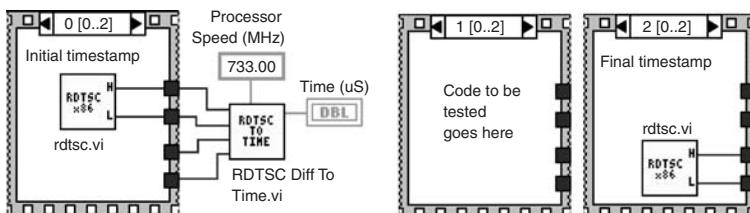
a safety loop that needs to reset a watchdog timer once per second to keep the process from being shut down. The safety loop is the real-time task, and the 1-kHz process will have to be preempted while the safety loop executes. To build a successful real-time system, you need to balance a limited number of real-time tasks with very close attention paid to how those tasks might interact or clash with one another. The remainder of your code should be tolerant of preemption; that is, it should tolerate much greater latency.

The next thing you need to know is the timing requirements that you have to meet. LabVIEW's built-in timing function uses the PC's real-time clock, which is based on milliseconds. To achieve timing loops faster than 1 kHz, you'll need to use hardware-timed acquisition, where a DAQ board provides the clock. This is not a problem, but you need to know in advance what your restrictions are so you can design for success. Be sure to leave enough spare time in your real-time task so that background tasks can execute. For example, a 10-kHz PID loop probably isn't going to do you much good if you can't send it a new set point.

Once you have your process timing requirements, you need to analyze your code or VIs and to determine how long each VI or collection of VIs takes to execute. After all your code is optimized and verified to fit within your timing requirements, you construct your program so that each time-critical piece of code can execute on time, every time.

### Measuring performance

LabVIEW's built-in VI profiler provides a good first indicator of VI execution time. But to help you get down to the real nitty-gritty of how many processor cycles a VI will take, National Instruments has also given us some tools to read the Pentium's built-in timestamp function. This allows you to get nanosecond-scale readings on execution time. The **RDTSC Timing** library includes a low-level VI (**RDTSC.vi**) that calls a CIN to read the CPU clock, and a utility to convert the resulting tick count to seconds (**RDTSC Diff To Time.vi**). There are also some examples that measure and display timing jitter. All you have to do is to insert your code to be tested. Figure 15.3 shows a very simple example.



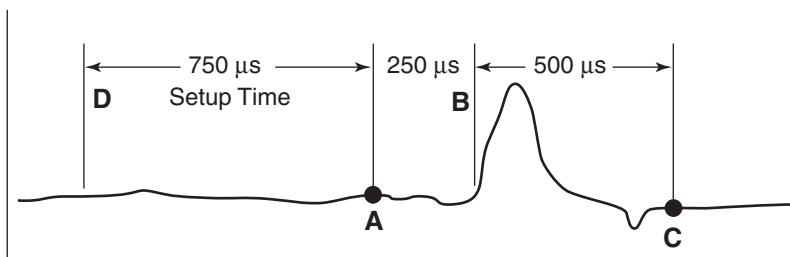
**Figure 15.3** The RDTSC library includes a VI to read the Pentium timestamp and convert it to microseconds. Use it for direct measurement of jitter and other small delays in your code.

Have you ever wondered what the real effect of that coercion dot was? This will tell you. You can find the RDTSC VIs on the Web at [www.ni.com](http://www.ni.com).

An oscilloscope is an essential tool on the hardware test bench, and it should be one of the first tools you turn to when analyzing real-time embedded performance as well. You can bring out many of the internal timing signals of your data acquisition card to an external terminal block like the SCB-68. A logic analyzer is another handy tool when most of the signals you're concerned about are digital. Looking at your timing signals is a great way to monitor the performance of your software and hardware.

We once had a data acquisition problem where we had to sample a photodetector immediately before and after a laser pulse (see the timing diagram in Figure 15.4). The laser fired at 30 Hz, and the first sample needed to be taken 250  $\mu$ s before the laser pulse and the second sample 500  $\mu$ s after the laser pulse. To provide this uneven timing, we programmed our DAQ card to sample the same channel twice each time an external scan clock pulsed. Then we set the interchannel delay on the DAQ card to 750  $\mu$ s. This meant that whenever the DAQ board received a trigger it would take the first sample and then wait 750  $\mu$ s before taking the second sample. By monitoring our board's timing signals we were able to verify exactly when each scan was beginning and when each sample was being taken.

Monitoring our DAQ card's timing signals also allowed us to see a “gotcha” of NI DAQ: When you use an external scan clock, the first sample is not taken until one interchannel delay after the input trigger. This meant that we were totally missing the event we were trying to see. But once we saw where our timing problem was, we were able to adjust our pretrigger input to the external scan clock so that the timing was just as specified. After the triggering was set correctly and the board was configured to store the data into a circular buffer, our



**Figure 15.4** Monitor AI CONVERT\* to see when each A/D conversion is taking place. We needed to sample a photodiode (A) 250  $\mu$ s before a laser pulse (B) and 500  $\mu$ s after the pulse (C). The DAQ card was configured to use an external scan clock trigger (D) with an interchannel delay of 750  $\mu$ s. An extra delay equal to the interchannel delay (750  $\mu$ s) was required by NI DAQ to set up the acquisition.

DAQ processing application only had to monitor the buffer for new data and act on it as it came in. You should use hardware triggering any time you can, even in LabVIEW RT. It will give you the most stable and repeatable system. It's also fully verifiable in that you can directly observe the signals externally.

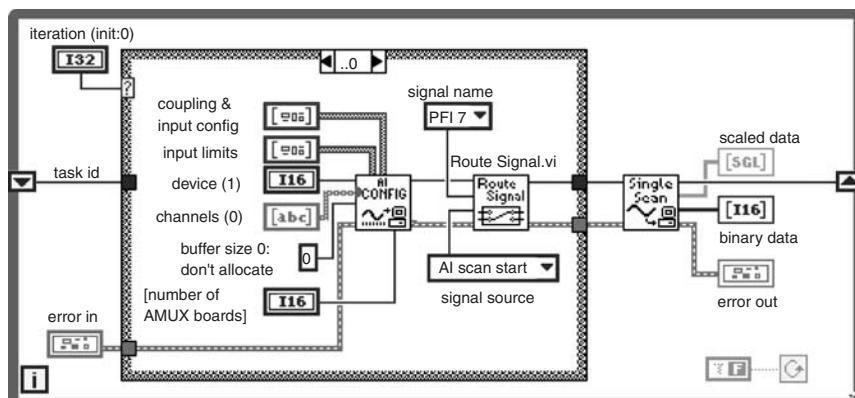
To view your board's internal timing signals, you need to use the **Route Signal VI** (available in the DAQ Miscellaneous function palette) to bring them out to a PFI pin on the connector where you can connect to them. You can wire this VI into your DAQ configuration routines and leave it there during development. It doesn't add any overhead to the data acquisition tasks, and you will find yourself relying on these signals whenever you need to verify timing. Figure 15.5 shows an example where the AI Read One Scan VI has been modified to include signal routing. Some of the key pin-outs to monitor on the E series cards are as follows:

**AI STARTSCAN** generates a 50-ns-wide, positive-going pulse at the beginning of each analog input scan. Use the Route Signal VI to connect the AI STARTSCAN signal to PFI 7.

**AI CONVERT\*** goes low for 50 ns at each analog/digital conversion. Use the Route Signal VI to connect AI CONVERT\* to PFI 2. This digital signal is useful to see when each analog/digital conversion is taking place in a scan.

**AO UPDATE\*** goes low for 50 ns when analog output circuitry is being updated. Route it to PFI 5.

Look at the documentation that comes with your DAQ card to verify your particular pin-outs.



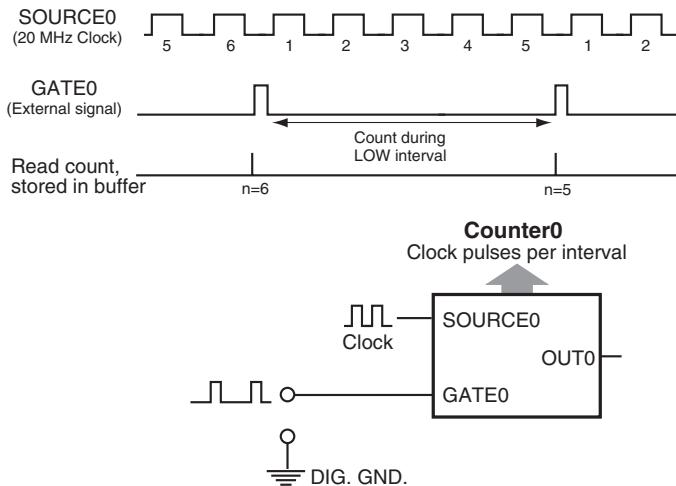
**Figure 15.5** AI Read One Scan modified to bring out the data acquisition card's internal timing signals. The configuration case has been altered to include the Route Signal VI. The signal being routed is AI STARTSCAN to PFI 7. Each time AI Read One Scan executes, PFI 7 will produce a pulse.

Almost any decent bench oscilloscope will let you see these brief transitions. If you're lucky and have a digital oscilloscope with a variable-persistence display, you can probably get a decent measurement of the software latencies, which will show up as jitter. But we'll show you an even better way to get an accurate timestamp of each and every transition by using something you're just as likely to have around: an extra DAQ card with a counter (Figure 15.6). There is an example VI called **Count Buffered Edges**. This VI uses one of the DAQ-STC counters on an E-series DAQ card to measure the period of a digital signal at the gate of one of your counters.

The counter measures the period of your signal by counting a signal connected to the counter's source and storing that count in a buffer each time it receives a signal at its gate. On the example VI, Count Buffered Edges, the counter's source is connected to the internal 20-MHz clock. This means the count increments every 50 ns. One of the features of buffered period measurement is that with each new period the counter is rezeroed, and you don't have to worry about the counter rolling over.

You can use the Count Buffered Edges VI to measure the performance of your real-time data acquisition tasks. Use the Route Signal VI when you configure your analog input or output to send the appropriate timing signal out to the connector. Then connect the output of that PFI pin to the gate of a counter on your extra DAQ board.

Now you can use this simple tool to "peek under the hood" of your real-time system and analyze your data acquisition tasks. Count Buffered

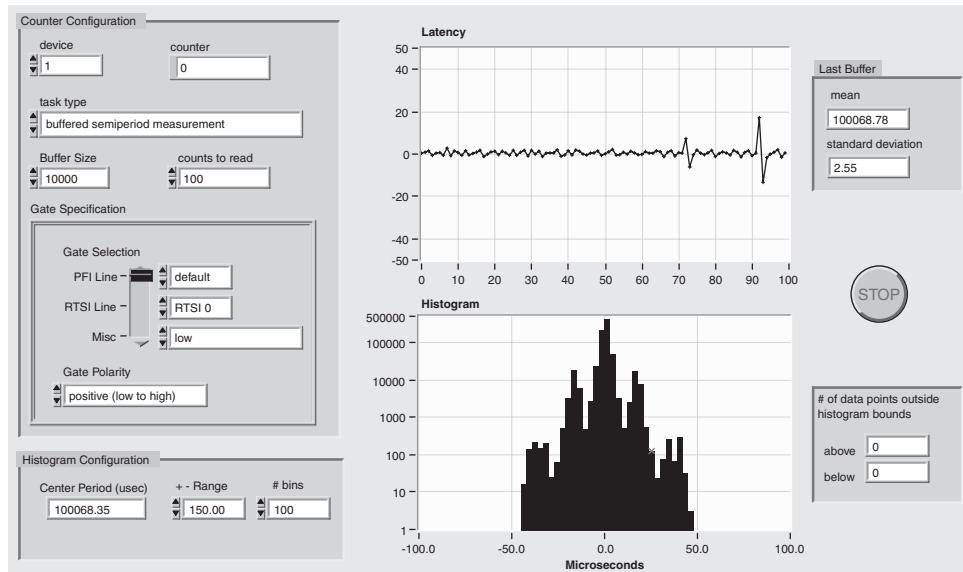


**Figure 15.6** A DAQ counter counts the 20-MHz internal clock, which is routed to the SOURCE input. Each time the counter's GATE is triggered, the total count is stored in a buffer. The counter is then rezeroed and counting continues.

Edges lets you see just how deterministic your loops are. The histogram plot keeps a running tally of latencies that you can use for long-term profiling to verify that your code is executing on time, every time.

Sometimes your measurements may seem very erratic, and this can be caused by noise on the digital lines. If you have a fast oscilloscope, you might be able to see noise or glitches that are triggering the counter. One way to cure the problem is by conditioning your signal with a one-shot, which is a pulse generator with a programmable pulse width. Set the pulse width to a value that's a bit less than the expected interval you're measuring, and the extra pulses in between will be ignored. There is a simple VI in vi.lib (vi.lib\DAQ\1EASYIO.llb\Generate Delayed Pulse.vi) that will let you use one of your counters as a delay generator. Set the delayed pulse width to be several tens of microseconds, connect the signal you want to buffer to the counter's gate, and monitor your cleaned-up signal at the counter's source.

If, for example, you want to look at AO UPDATE\* to see when the analog output circuitry on your real-time card is being updated, you connect PFI 5 to counter 0's GATE. The slightly delayed buffered pulse will be at counter 0's OUT. Buffering your signals in this way will make them easier to see, and it will also eliminate some of the noise that might be giving you false readings. No extra software overhead is added to your program other than the time required to initialize the counters. Figure 15.7 shows the results obtained when a digital line was toggled



**Figure 15.7** Screen capture from the Count Buffered Edges VI after toggling a digital line every 100 ms for 20 h. The histogram of latencies shows that there were no latencies greater than 50  $\mu$ s.

every 100 ms by writing to a digital port in a LabVIEW RT For Loop. The peak jitter was only 50 microseconds after 20 h of operation.

Once you've streamlined each section of your code and verified that it can meet its deadlines, it's time to start putting everything together to make a complete application. This is when we need to know how the LabVIEW execution system works and how your operating system affects execution.

### Shared resources

If unexplained timing glitches occur when you run your code, then you may have the problem of two sections of code competing for a shared resource. The quickest way to lose deterministic execution of your code is to have multiple tasks in competition.

A **shared resource** is a software object that cannot be accessed simultaneously by multiple threads. Examples of shared software resources are single-threaded device drivers such as NI DAQ, global variables, shared subVIs that are not reentrant, synchronization code (semaphores, queues, occurrences, rendezvous), networking protocols, file I/O, and LabVIEW's memory manager.

To protect a shared resource such as NI DAQ calls, LabVIEW places a **mutex** around it. A mutex, or mutual exclusion, is a software mechanism to prevent two threads from accessing a shared resource at the same time. You can think of a mutex as a lock on a door that has only one key. When your program uses NI DAQ to access a data acquisition board, it must first check to see if NI DAQ is in use. If NI DAQ is available, your program gets the "key," unlocks the door, and enters the protected area, and the door is locked behind it. No other program can use the resource until the door is unlocked and the key is returned.

For example, you can write a program in regular LabVIEW that uses two data acquisition loops running in parallel; maybe one is analog input and the other is digital I/O. Because you're not concerned about or monitoring the real-time behavior of these DAQ loops, everything appears to be fine—only the average time of the loops is important. But if you do this in a real-time execution system where you are paying close attention to deterministic timing, you will see how each DAQ task is interfering with the other as they compete for the shared resource.

If two tasks competing for a shared resource are at different priorities, you can get a condition known as **priority inversion**, where a high-priority task has to wait for a low-priority task to finish. If a low-priority VI has the key to the shared resource, that VI cannot be preempted by a higher-priority VI. The high-priority task will have to wait for the low-priority task to finish. One method used by the

RTOS to resolve a priority inversion, known as *priority boosting*, is to temporarily raise the priority of the task using the shared resource to **time critical** until the shared resource is released. Then the low-priority task is reduced back to its original priority and the higher-priority task is free to run.

A shared resource that you don't usually think about is LabVIEW's memory manager. LabVIEW manages memory dynamically behind the scenes, and it is so transparent that you usually never even have to think about it. But each time LabVIEW manipulates a region of memory, it uses a mutex to protect the region so that other parts of your application don't interfere halfway through the process and mess things up. For example, this means that a low-priority subVI that is building an array in a different execution system from our time-critical loop can have a dramatic effect on deterministic behavior. A solution to this particular problem is to preallocate all your arrays at the beginning of the program and avoid resizing them. While you can't totally avoid memory management, you should at least follow some of the general guidelines in Application Note 168, *LabVIEW Performance and Memory Management*.

### Multithreading and multitasking

Ever since version 1.0, LabVIEW has given us the ability to execute pieces of our program in parallel. LabVIEW's dataflow paradigm makes it easy for us: All we have to do is to wire up two parallel While Loops and presto! It works! LabVIEW performs this magic by **multitasking**, whereby each section of code is given a small slice of time to run on the processor before it is pulled off and another section is allowed to run. The LabVIEW execution engine controls the multitasking in a cooperative manner. To a great degree, the multitasking that takes place is transparent to us; our code apparently executes in parallel. In the latest versions of LabVIEW, we still have the cooperative multitasking of the LabVIEW execution engine, but it has been enhanced by the addition of **multithreading**.

Beginning with LabVIEW 5.0, when you write a LabVIEW application on a multithreaded OS and push the Run button, your compiled program is split up into multiple copies of the LabVIEW execution system. Now, in addition to LabVIEW executing portions of your diagram in parallel, you have the OS running multiple copies of the LabVIEW execution system in parallel. These multiple systems running in parallel are a powerful programming concept that is made easy by LabVIEW.

There are at least two references you should consult with regard to multithreading and execution of VIs in LabVIEW. Application Note 114, *Using LabVIEW to Create Multithreaded VIs for Maximum*

*Performance and Reliability*, is available from the LabVIEW help system (under printed manuals). If you have a copy of LabVIEW 5, the *G Programming Reference Manual* has a helpful chapter entitled “Understanding the G Execution System.”

The default configuration of LabVIEW has one copy of the execution system for each of four priorities: *time critical* is the highest, then *high*, *above normal*, and *normal*. The default also calls for one copy for each of five categories: *standard*, *instrument I/O*, *DAQ*, *other1*, and *other2*. (The category names are arbitrary. That is, the *DAQ* category can be assigned to any task, not just *DAQ* operations.) An additional priority, *background*, is not enabled by default, but you can enable it and change the number of threads in each execution system by using the **Threadconfig VI** located in /vi.lib/utilities/sysinfo.llb (Figure 15.8). Each of the 20 default copies of the LabVIEW execution system runs a set of VIs that you assign to it as its thread runs on the processor. The LabVIEW threads and priorities that you assign to your VIs are mapped directly to priorities within the OS. The OS assumes complete responsibility for which thread gets the processor.



**Figure 15.8** You can set how many threads are assigned to each LabVIEW execution system with the Thread Configuration VI. Find it in labview/vi.lib/utility/sysinfo.llb.

A thread can be in one of three basic states:

**Running.** The processor is running the instructions that make up this thread.

**Ready.** Some other thread is running, but this thread could run if it were given access to the processor.

**Blocked.** The thread is waiting for an external event such as an interrupt or a timer. Even if the processor were available, this thread could not run.

The RTOS used by LabVIEW RT uses a combination of **round-robin** scheduling and **preemptive** scheduling. In round-robin scheduling, each thread that is ready to run gets a slice of time to execute. In preemptive scheduling, the operating system takes control of the processor at any instant. That's a way for higher-priority tasks to get preferential access to the processor, thus improving the response time of specified tasks. The RTOS will give the processor to the ready thread with the highest priority, but if there are multiple threads with the same priority, the RTOS will give each thread a small slice of time. Each thread will run until it blocks (waiting for a timer or interrupt), a higher-priority thread unblocks, or the RTOS pulls it off to run another thread at the same priority. Any work that the thread did not complete will have to wait until the next time it can run.

The one exception to the time-slicing feature of each execution system occurs when a VI is set to run at **subroutine priority**. A subroutine VI will always run to completion within its execution system. A thread running a subroutine VI can still be preempted by the RTOS, but no other VI running in the same execution system as a subroutine VI will be able to run until the subroutine VI is finished.

### Organizing VIs for best real-time performance

To make best use of the parallel execution systems, consider which VIs are logically grouped together and keep them within the same execution system. If you have a communications loop that you use to pass data between the RT system and the host computer, then place that loop and all related VIs into one thread or execution system. Set the thread to a low priority, such as background or normal, and let the RTOS take responsibility for running this thread when higher-priority threads are sleeping. By letting the RTOS take care of the hard work for us, we can maximize the efficiency of our application.

Place your real-time task in a time-critical thread, and it will run before any other thread. When the time-critical priority thread is ready

to run, then the scheduler will preempt or remove whatever thread is running from the processor and the time-critical priority thread will run until it blocks. Be sure that you never have more than one time-critical thread, or else the RTOS will implement round-robin scheduling between the time-critical threads and you will lose your deterministic performance.

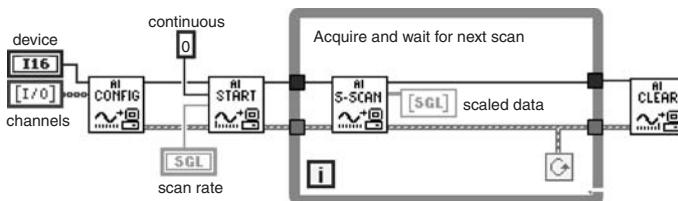
There are several things that you should watch out for when assigning priorities to your VIs in LabVIEW RT that are different from normal LabVIEW. A VI set to time-critical priority must have a *sleep* function somewhere inside its main loop. The only time low-priority threads get a chance to run is when the time-critical thread blocks or sleeps. A time-critical thread that never sleeps will starve out all other threads.

There are at least two ways to create a sleep function. First, you can use the Wait For Next ms Multiple or the Wait ms function inside its main loop. Either of these timers can wake up the loop once every integral number of milliseconds. While the loop is sleeping, the RTOS will schedule lower-priority threads until it is time to wake up the loop again.

Second, you can use the DAQ function AI Single Scan. You can configure a hardware-timed acquisition external to your time-critical loop and call AI Single Scan to read each scan as it becomes available (Figure 15.9). This is counterintuitive because normally calling a DAQ function in this way causes the loop to use all the processor time until the function returns. NI DAQ has been modified for LabVIEW RT so that this one VI provides its own sleep function.

When a loop in a time-critical priority VI is put to sleep, the entire thread is put to sleep. No other loops or VIs in that thread will run until the thread wakes up. This means that you cannot have two parallel loops running at different rates inside the time-critical thread. If you need to have multiple tasks operate in the time-critical loop, then you need to implement a scheduler to control execution.

In addition to the various execution systems that manage the sub-VIs, another execution system is used for the user interface in



**Figure 15.9** AI Single Scan provides its own sleep function. The VI shown here would typically be set to time-critical priority in LabVIEW RT. This is not standard practice in regular LabVIEW.

regular LabVIEW. But in LabVIEW RT this user-interface thread becomes a TCP/IP communication thread between the RT system and the real user interface running on the host development system. That's how we remove the burden of a user interface from the RT processor.

Having a large number of execution systems allows us to make more efficient use of the processor but at the expense of speed and determinism. The more execution systems you use, the more likely it is that you will need to share data among two or more execution systems. LabVIEW takes care of the details of this for you, but there is still some overhead involved in passing data back and forth.

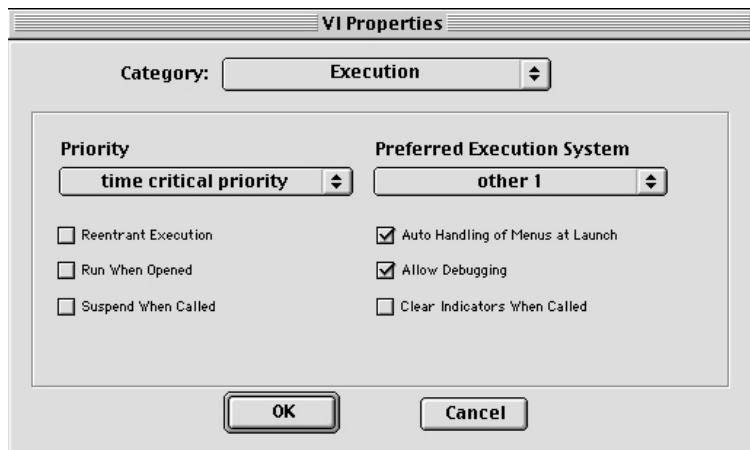
There are several options for sharing data between a real-time loop and a lower-priority loop. If you don't need to collect every data point, then you can use a LabVIEW 2-style shift register global set to subroutine priority with **skip if busy** enabled. When you drop a VI set to subroutine priority on the block diagram, an extra menu item shows up when you command click on the subVI: "Skip Subroutine Call If Busy." The real-time loop will skip the VI if the VI is in use by a lower-priority thread. You may miss a few data points, but you can build whatever functionality you need into your global. To avoid missing data points, you can build a queue or buffer to transfer data to the lower-priority loop.

A good application for all that we've discussed so far is point-by-point control loop I/O. In these situations, you need to make a measurement, do a computation (such as PID), and then generate an output with minimal delay or timing jitter. You would build such a VI in the manner shown in Figure 15.9 and set the VI's priority to time critical. It should run as a separate top-level VI, and it should be the only time-critical VI in the system. You can exchange data such as the setpoint, measurement, and status information via global variables or queues. We'll see such an example later.

### Context switching adds overhead

It's important to choose carefully which thread your VIs are assigned to. The actual names of the execution systems are not important; that is, DAQ VIs do not need to run in the execution system named DAQ. What is important is that you avoid placing subVIs that are called repeatedly in separate execution systems from their caller. You assign a VI to run in a specific thread using the Execution options from the VI Properties dialog (Figure 15.10).

A **context switch** occurs each time the OS pulls one thread off the CPU and puts another thread on. When the OS swaps threads, all the stack information and all the instructions in the CPU cache need to be swapped out to main memory. Then all the new thread's information



**Figure 15.10** You can assign a VI to go in a specific execution system by using the Execution options from the VI Properties dialog. The default setup for each VI is normal priority in the same execution system as the caller. It's the setting you almost always want for subVIs.

is loaded into the CPU so that the thread is in the same state it was in before it was removed. This can take a significant amount of time. Any time you force the operating system to perform a context switch by calling a subVI in another execution system, you cause delays that will affect how efficiently your code executes. If the subVI needs to be in a separate execution system, then consider making it into a top-level VI, just as you would for a time-critical VI.

Consider what happens when you have a top-level VI set to run in the standard execution system and a subVI set to run in the DAQ execution system. Each time that subVI is called, the thread containing the top-level VI will be removed from the processor and the thread containing the subVI will run. When the subVI is finished, another context switch occurs as the OS switches back to the top-level VI. If the subVI is in a loop that runs very frequently, the extra overhead will be easy to observe. This shows why you should not arbitrarily assign execution systems to subVIs.

When a subVI is set to execute in the same execution system as the caller, it will inherit the caller's priority and run in whatever execution system it was called from. That is, if you have a top-level VI set to time-critical priority in the standard execution system and it calls a subVI set to normal priority and the same execution system as the caller, there will not be a context switch from time-critical to normal priority. The subVI will have its priority elevated to the same priority as the calling VI.

## Scheduling

**Timed structures** simplify the way you schedule real-time execution order. Efficient scheduling and the mechanisms used to enforce it used to be a complex art form driven by which VI has the highest priority and the earliest deadline. Now you just place down a Timed Loop, give it a priority and a period, and off you go. If you need a parallel task at a different loop rate and a higher priority, that's no problem! Once again LabVIEW makes something incredibly easy that was terribly complicated.

### Timed structures

There are three structures with built-in scheduling behavior: **Timed Loops**, **Timed Loop with Frames**, and **Timed Sequences**. All three provide a way to schedule real-time program execution. Each Timed structure runs in its own thread at a priority level just beneath time critical and above high priority. Timed Loops do not inherit the priority or execution system of the VI in which they run. This means that a Timed structure will stop any other code from running until the structure completes. LabVIEW's Timed Structure Scheduler runs behind the scenes and controls execution based on each Timed structure's schedule and individual priority. Each Timed Loop's priority is a positive integer between 1 and 2,147,480,000. Priorities are set before the Timed Loop executes, through either the configuration dialog or a terminal on the Input node. Figure 15.11 shows the configuration dialog for

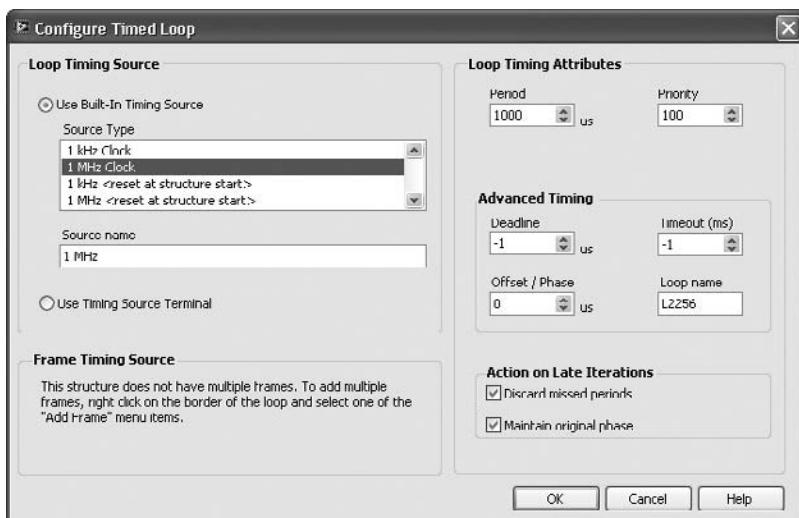
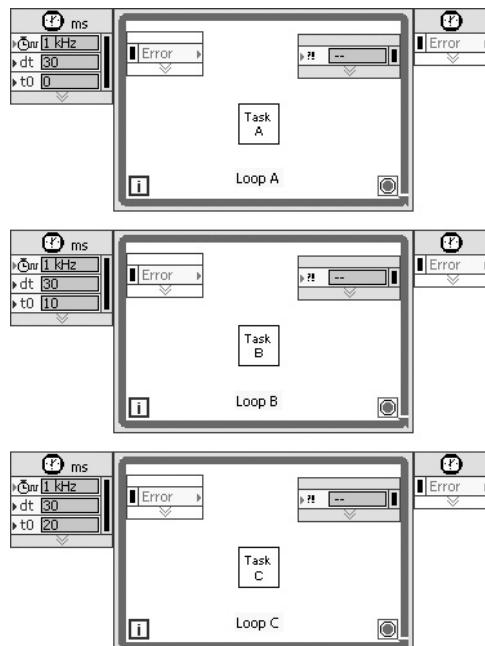


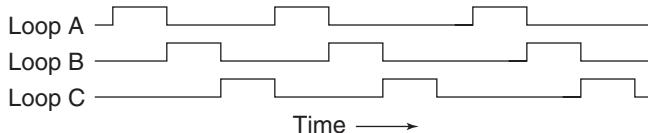
Figure 15.11 Configuration dialog for a Timed Loop. Timed Loops on RT systems have a 1-MHz timing source available.

a Timed Loop. Priority can be adjusted inside the Timed Loop through the right data node, so you can dynamically adjust performance if needed. Timed structures at the same priority do not multitask between each other. The scheduler is preemptive, but not multitasking. If two structures have the same priority, then dataflow determines which one executes first. Whichever structure starts execution first will finish before the other structure can start. Each Timed Loop runs to completion unless it is preempted by a higher-priority Timed Loop, a VI running at time-critical priority, or the operating system.

Figure 15.12 illustrates a round robin schedule implemented with 3 timed loops. All 3 loops use the 1 kHz system clock with a period (dt) of 30 ms. Loop B's start (t0) is offset by 10 ms from Loop A and Loop C is offset from Loop A by 20 ms. The timeline in Figure 15.13 shows each loop's time slice provided each task finishes in its allotted 10 ms. If one of the tasks took longer than 10 ms to complete the Timed Structure Scheduler would adjust execution based on priority, and the other loops' configuration options for late iterations. Any loop with a higher priority will preempt a lower priority loop. Once the higher priority loop has finished the preempted loop will get a chance to finish.



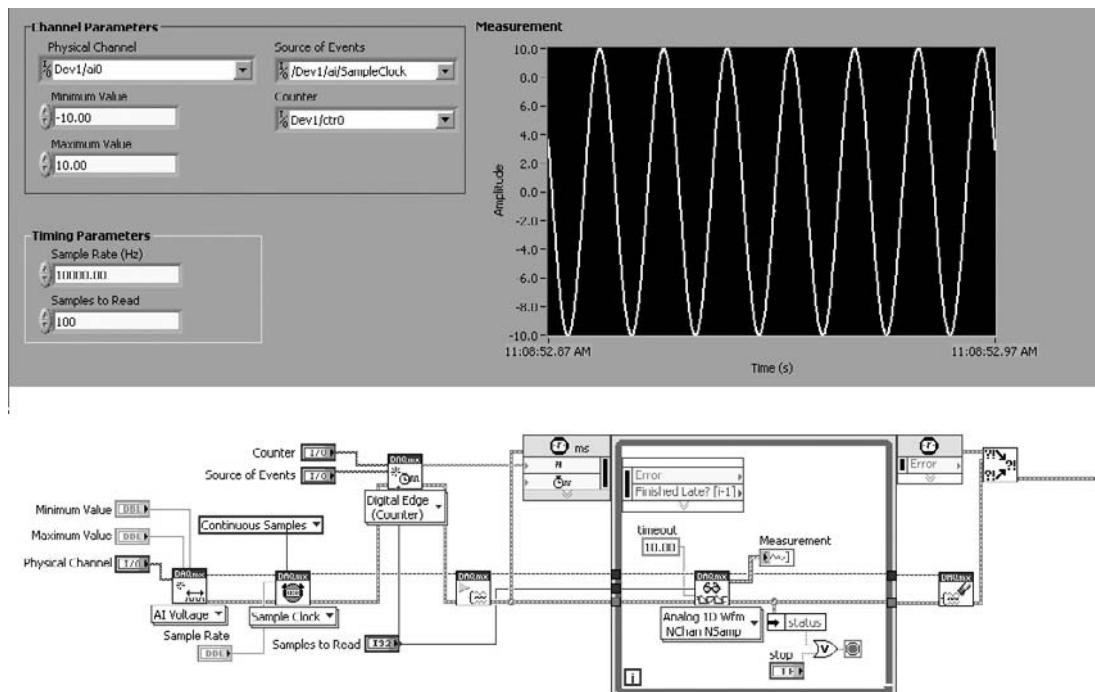
**Figure 15.12** Three Timed Loops with the same period, but offset in phase (t0). They will run one after the other as shown in Figure 15.13.



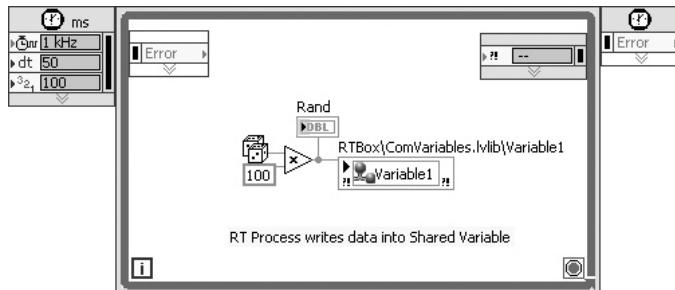
**Figure 15.13** Time line for the three Timed Loops in Figure 15.12. All loops have the same period, but are offset in phase.

Late starts by timed loops are governed by the selected actions on late iterations as shown in Figure 15.11. The default selections will cause a timed loop with a late start to forgo execution until its next scheduled timeslot. Deselecting “Discard missed periods” causes the timed loop to try to make up any missed iterations. This can have unexpected consequences, so use with caution and always verify that your application is doing what you expect.

Timing sources for Timed structures can be based on your computer’s internal timer or linked through DAQmx (Figure 15.14) to a hardware-timed DAQ event with **DAQmx Create Timing Source.vi**. You can even chain multiple timing sources together with **Build Timing**



**Figure 15.14** Hardware-timed acquisition and control. Timed Loops can be synchronized to DAQmx events.



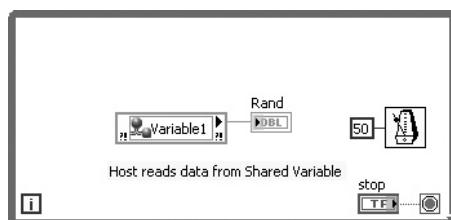
**Figure 15.15** Shared variable is used to communicate between the RT process and the host PC (Figure 15.16).

**Source Hierarchy.vi.** This could allow you to execute based on time or, with the appropriate hardware, on a change on any configured digital line.

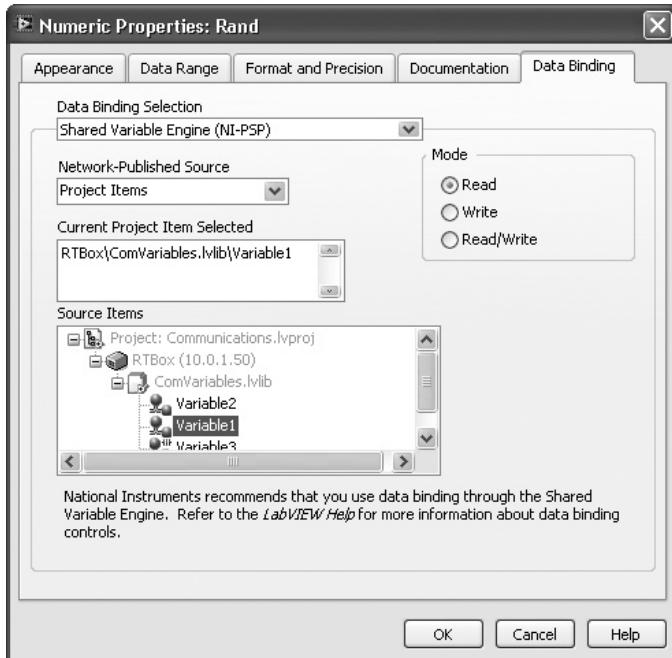
## Communications

LabVIEW has several technologies to make exchanging data transparent between distributed real-time targets and a host PC. The easiest to use is the **Network-Published Shared Variable**. Figures 15.15 and 15.16 show all the code you need to communicate between a real-time process and a host PC. Shared variables communicate using National Instruments' **Publish-Subscribe Protocol (NI-PSP)** with a **Shared Variable Engine (SVE)** that hosts the shared variables on the network. When you write to a shared variable, LabVIEW sends the data to the SVE which then **publishes** (sends) the data to all **subscribers**. You can directly link front panel control and indicators to shared variables (Figure 15.17). This is really easy, but totally violates dataflow.

Shared variables may not be right for every application, especially if you need to communicate with programming languages outside of LabVIEW. For those applications consider the Real-Time Communication Wizard (Tools >> Real-Time Module >> Communication Wizard).



**Figure 15.16** Host PC reads data from the shared variable.



**Figure 15.17** Data binding allows you to bind front panel objects to shared variables without programming.

The Communication Wizard converts an existing real-time VI into a client-server architecture. Controls and indicators in the time-critical code are replaced with real-time **FIFOs (First-In, First-Out)** that transfer data to a parallel lower-priority communication loop. An additional communication VI is created that your host uses to talk to the time-critical process. Programmatically it's considerably more involved than the shared variable approach, but it's there if you need it.

## Bibliography

- Application Note 200, *Using the Timed Loop to Write Multirate Applications in LabVIEW*, www.ni.com, National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2004.
- Hays, Joe: "Advanced Real-time Programming Techniques," NIWeek 2000, Advanced Track Session 2D, available from www.natinst.com/niweek.
- Li, Yau-Tsun Steven, and Sharad Malik: *Performance Analysis of Real-Time Embedded Software*, Kluwer Academic Publishers, Boston, 1998.
- Simon, David E.: *An Embedded Software Primer*, Addison-Wesley Longman, Reading, Pa., 1999.

*This page intentionally left blank*

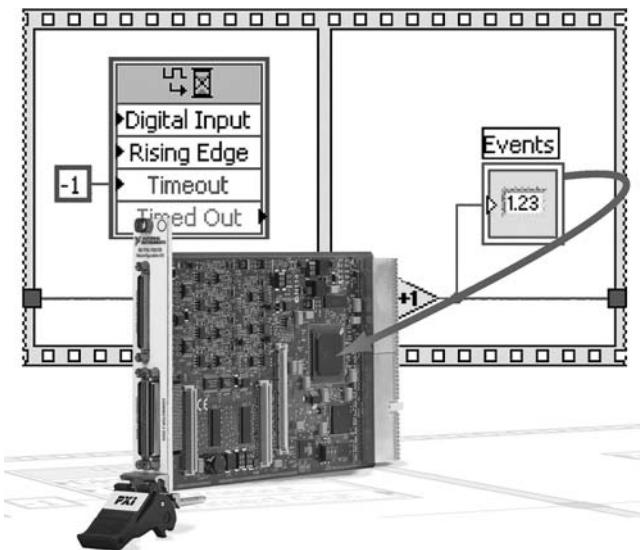
## LabVIEW FPGA

LabVIEW FPGA takes real-time programming to the hardware level. Although LabVIEW RT is a deterministic real-time environment, it is still software running on a microprocessor, and any microprocessor has only a limited number of clock cycles available to share among all the running processes. LabVIEW RT's deterministic behavior means that you can accurately predict when your code will run; but as your application grows and you start running more processes, you have to begin making tradeoffs that affect determinism.

LabVIEW FPGA applications are not constrained by processor or operating system overhead. Code is written in parallel, is mapped directly into parallel hardware registers, and runs in parallel. With LabVIEW FPGA you can write massively parallel hardware-timed digital control applications with tight closed-loop rates in the tens of megahertz.

### What Is an FPGA?

FPGA is the acronym for field-programmable gate array. The technology has been around for a while, but only recently have the costs come down and the tools improved enough that FPGAs have become commonplace. An FPGA is essentially a 2D array of logic gates with software programmable interconnections. FPGAs range in size from devices with tens of thousands of gates to devices with millions of gates and multiple embedded microprocessors. The number of gates is really just a marketing number. Each FPGA vendor combines those gates to create its own unique architecture. The basic programming block of the Xilinx parts used by National Instruments is defined as “slices.” Each slice contains two storage elements, arithmetic logic gates, large



**Figure 16.1** PXI 7831R is National Instruments' first reconfigurable intelligent DAQ card. User code is downloaded into the 1 million gate LabVIEW programmable FPGA. (*Photo courtesy of National Instruments.*)

multiplexers, a fast carry look-ahead chain, a horizontal cascade chain, and two function generators. The function generators are configurable as 4-input lookup tables, 16-bit shift registers, or 16-bit memory. As you develop and compile your applications, look at the compilation summary and keep an eye on how many slices your application uses. The 1 million gate XCV1000 FPGA used on the 7831R (Figure 16.1) contains 5120 slices. Try out different approaches to a problem and see how each approach affects the number of slices used. Programming with LabVIEW FPGA doesn't require knowledge of a hardware definition language such as VHSIC hardware description language (VHDL) or the underlying hardware; but understanding what's underneath the hood and following a few basic guidelines we cover in this chapter will help you write cleaner, faster code.

An FPGA is a blank slate you can use to create almost any logic or control circuit you want, and once the programmable interconnections between the gates are made, your software becomes hardware. They can be configured to precisely emulate discrete logic systems, or even replace obsolete components. You can find FPGAs serving as the timing and triggering logic on many plug-in cards, including National Instruments' DAQ cards. Unfortunately the FPGA on the standard E-series and M-series DAQ cards is not large enough to allow user-programmable

code, but hopefully in the future all NI's DAQ cards will have user-programmable logic. A presentation at NIWeek04 pointed out the major shift in design philosophy between the LabVIEW FPGA programmable DAQ cards and traditional DAQ cards and drivers (NI DAQ or DAQmx). System designers of traditional DAQ cards design a flexible hardware personality into the card and engineer a full-featured driver to expose the hardware as the designer intended. It's that "as the designer intended" part that causes trouble. The designer couldn't anticipate your application or its exotic analog and digital triggering conditions. LabVIEW FPGA and the RIO cards give you, the end-user/programmer, a wide-open piece of hardware and a lightweight driver to handle communications between the card and the PC. Whatever you want to do on the card is in your hands. This is a revolutionary change.

## LabVIEW for FPGAs

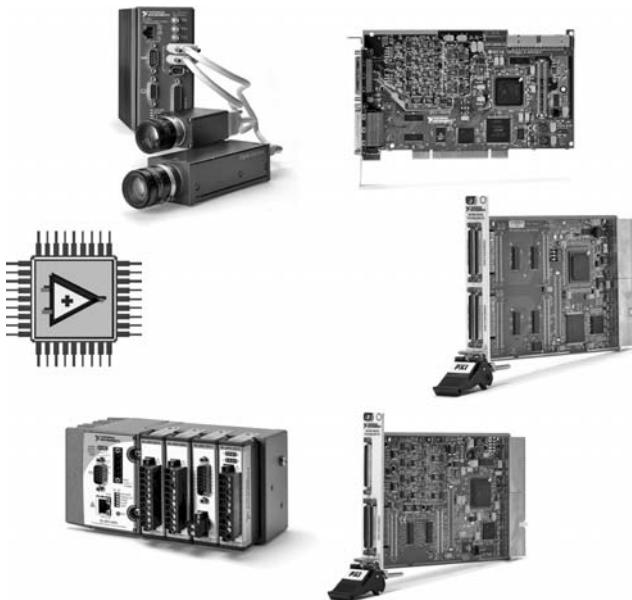
It's easy to visualize and draw out parallel operations in LabVIEW, and this maps really well into logic on an FPGA. LabVIEW FPGA gives you the ability to quickly prototype right down at the hardware level and to easily build low-level hardware interfaces with LabVIEW. This is a huge advantage for digital design and development teams. Richard's group at Sandia builds custom hardware for telemetry. One of the biggest challenges has been to find a way to interface custom digital hardware to a desktop PC for test and development. Now adding intuitive, graphical interfaces to low-level communication protocols such as SPI, I2C, and custom serial and parallel busses is easy.

## RIO hardware platforms

National Instruments' RIO (Reconfigurable I/O) LabVIEW programmable hardware (Figure 16.2) wraps a high-level interface, and a lot of I/O goodies, around a Xilinx FPGA. Programming the FPGA is easy. You have almost infinite control over the asynchronous I/O. By asynchronous I/O we mean the input/output lines can be used independently of each other; however, you can synchronize them if you want. You can use the digital-to-analog converters (DACs) for arbitrary pulse and waveform generation. Complex triggering with digital or analog triggers is no problem. The DIO lines can be used for counters, timers, and frequency dividers; just about anything you want to do with standard logic is possible.

## Plug-in cards

National Instruments currently has seven R-series intelligent DAQ cards varying in form factor, size of FPGA, and number and type of

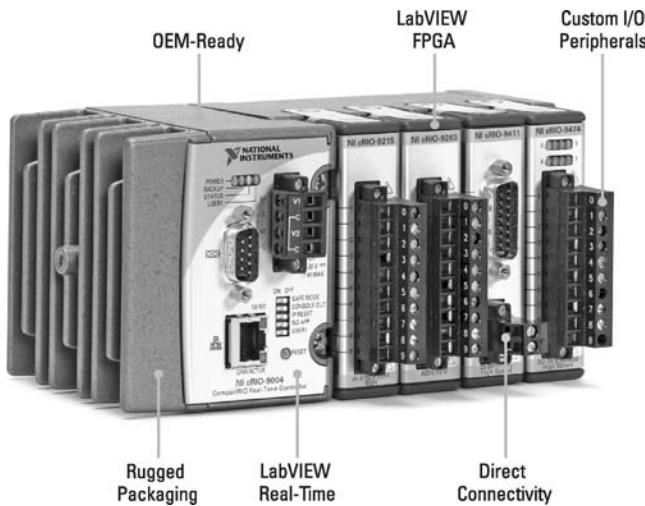


**Figure 16.2** LabVIEW FPGA reconfigurable hardware includes intelligent DAQ cards, timing and synchronization modules, Compact Visions systems, and the CompactRIO Programmable Automation Controller. (Photo courtesy of National Instruments.)

I/O lines. All the cards are designed to let you create your own processing algorithms that can run at rates up to 40 MHz. You can configure digital lines as inputs, outputs, counter/timers, pulse-width modulator (PWM) encoder inputs, or custom communication busses. On cards with analog capabilities you can have up to eight simultaneous 16-bit analog inputs at rates up to 200 kHz, and up to eight simultaneous 16-bit analog outputs at rates up to 1 MHz. The more we use LabVIEW FPGA, the more we're convinced that all DAQ cards should be user-programmable. All the trigger lines and PXI local bus lines are exposed on the RIO hardware, and you may use any triggering scheme, analog or digital, you desire. Because the input, processing, and control output are all embedded into the FPGA fabric, you will find that the RIO plug-in cards have exceptional performance for single-point I/O control applications.

### CompactRIO

CompactRIO or cRIO combines a LabVIEW RT controller for advanced floating-point analysis and control with an intelligent FPGA backplane for hard real-time timing, synchronization, and control. A cRIO system



**Figure 16.3** CompactRIO is a compact 3.5-in × 3.5-in × 7-in industrial controller with a LabVIEW RT processor and LabVIEW FPGA programmable backplane. (Photo courtesy of National Instruments.)

consists of an embedded real-time controller, a four- or eight-slot chassis with a LabVIEW FPGA programmable backplane, and a mix of industrial I/O modules (Figure 16.3).

### Timing and synchronization

Advanced timing and synchronization applications can use National Instruments' PXI 6652 and 6653 control modules to control timing across multiple PXI chassis for almost unlimited customization. In addition to synchronizing multiple chassis, each controller can drive the STAR trigger line and the TTL triggers on the PXI backplane. Typical uses for these LabVIEW FPGA programmable cards include advanced timing, triggering, and user-defined delay applications.

### Compact Vision

National Instruments' Compact Vision system is a rugged industrial vision system that uses a LabVIEW programmable FPGA for complex timing and synchronization. Each system has three FireWire camera ports, a VGA port, an Ethernet port, 15 digital inputs, and 14 digital outputs. You can program the digital lines with LabVIEW FPGA for quadrature encoders, generating strobe pulses for the vision system, synchronizing to external stimuli, or controlling relays and other actuators.

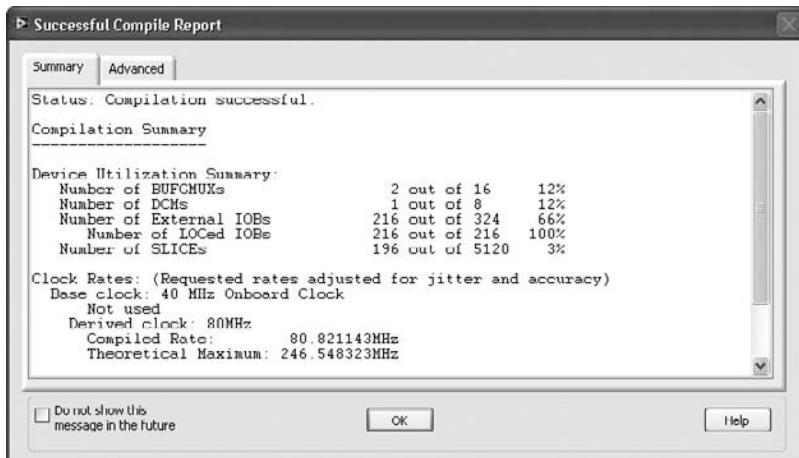
## Application Development

There is no single definitive programming method for LabVIEW FPGA, but there are some significant differences between LabVIEW on a desktop and LabVIEW code running on an FPGA. Here is a short list, which we'll expand on throughout the chapter:

- **Local variables.** Some things that are bad on the desktop, such as local variables, are good on FPGA. A local variable in desktop LabVIEW is inefficient and results in extra copies of the data everywhere it is used. On FPGA a local variable is a hardware register that can be efficiently accessed from multiple locations. Race conditions can still occur if you are not careful, so be sure to have one writer and many readers.
- **SubVIs.** SubVIs are efficient chunks of reusable code on the desktop, but they can have unexpected consequences on the FPGA. Unless a subVI is marked reentrant, there will be only one copy of the subVI mapped into hardware. A single copy of a VI in hardware may be what you want if you need to Read–Modify–Write a shared piece of data in multiple loops, but it will also mean that the loops have a hardware dependency on one another. This can cause unexpected jitter and may even destroy an expected real-time response.
- **Parallelism.** A common pitfall of LabVIEW programmers new to LabVIEW FPGA is to use the same logic flow they used on the desktop and to ignore the physical parallelism exposed by the FPGA. LabVIEW's parallel dataflow paradigm maps really well into FPGA programming because any node that has all its inputs satisfied will execute. This lets you easily write parallel control loops that actually run in parallel. And because your code is mapped into hardware registers, the number of deterministic parallel tasks is only limited by the size of the FPGA and the efficiency of your code.
- **Flat hierarchy.** Even though your application may have a hierarchical structure to it, it is compiled and mapped onto a “flat” two-dimensional array of logic without any hierarchy at all. Avoid excessively complicated logic with many nested Case structures because this maps very poorly onto the FPGA and consumes too many resources. Minimizing resource usage is a good thing to practice because no matter how big the FPGA is, sooner or later you'll write a program that is bigger.

## Compiling

Compiling for the FPGA can take some time and is the only painful part of programming with LabVIEW FPGA. When you compile for your FPGA target, LabVIEW converts your block diagram into VHDL and



**Figure 16.4** Successful Compile Report from LabVIEW FPGA. The number to watch is the number of slices used. In this case 196 slices out of 5120 were used.

then uses the Windows command line to call Xilinx's tools. The end product is a netlist prescribing how the hardware is interconnected. The compile process can take a long time, many tens of minutes on a PC with limited RAM. On a dual-Xeon machine with 2-Gbyte RAM an average compile is 3 min, and the longest has been 20 min. If you're developing on a PXI controller, set up a fast PC with a lot of RAM as a compile server. You'll save a lot of time.

Figure 16.4 shows a compile report that is returned after the project has successfully compiled. Here's a quick definition of terms:

- **BUFGMUX.** BUFG is a buffer for a global clock and MUX is a multiplexer. As you add derivative clocks you will consume BUFGMUXes.
- **DCM.** Digital Clock Manager is used for clock multiplication. In this project we derived an 80-MHz clock from the 40-MHz onboard clock.
- **External IOBs.** IOB is an input/output block. The XCV1000 on the 7831R has 324 IOBs, 216 of which are used.
- **LOCed IOBs.** These are located IOBs, or constrained to a fixed location.
- **Slices.** This is the most important number to watch as you optimize your code. The fewer the slices used, the more efficient your code is and the more you can do in a fixed FPGA space.
- **Base clock.** This is the fundamental onboard clock. You can derive faster clocks by using the 40-MHz reference. In Figure 16.4 we compiled our application using an 80-MHz clock.

- **Compiled rate.** You can compile your code to use clocks of 40, 80, 120, 160, and even 200 MHz, but the faster you go, the more jitter you introduce. Additionally, the digital I/O of the XCV1000 is slew-rate-limited at 20 MHz. The safest practice is to use 40 MHz as your base clock unless you really need some section of code to run faster.

As you develop applications for the FPGA, take time during development to compile and notice how many slices your application consumes. You can even cut and paste the compile report onto the block diagram or into the history report to keep a running record of the application size. Excessively convoluted logic consumes valuable resources, can cause unnecessary latency, and may even cause your code not to compile. To get maximum performance and the best use out of the FPGA, keep the code on the FPGA simple and clean.

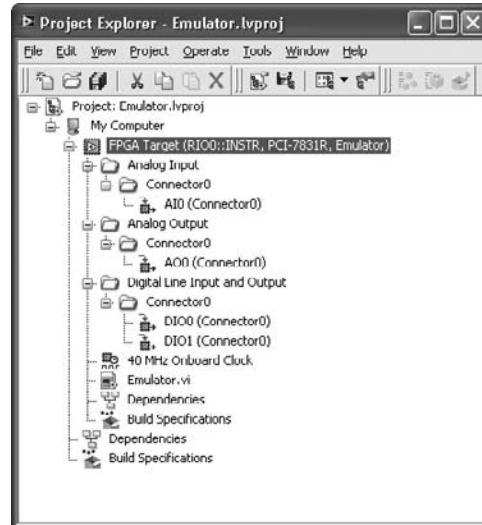
## Debugging

LabVIEW has some great debugging tools, but unfortunately they aren't available for compiled code on the FPGA. However, you can use LabVIEW's graphical debugging tools, including probes and execution highlighting, when you use the **Emulation** mode. To use the emulator, right-click on the FPGA Target in the project folder and select **Target Properties**.... I/O in Emulation mode can use randomly generated data, or you can use the actual I/O hardware of your FPGA target. This is a great way to do some rapid prototyping and debugging since VIs in Emulation mode do not have to go through the long compilation process. Of course there is no real-time response in Emulation mode since your VI is battling every other process on the host computer for CPU time. To get hard real-time response, you have to compile and run down on the hardware where it is much harder to peek into a running process. See Figure 16.5.

Place digital outputs at strategic points in your code, and toggle the digital line each time that section of code executes. Each loop is a potential point you may want to look at. Toggling a digital output once each loop iteration is a great way to see what is happening. Once your VI is compiled and running, you can connect an oscilloscope to these digital watch points to get a real-time look into your process. You might be surprised at what you find out.

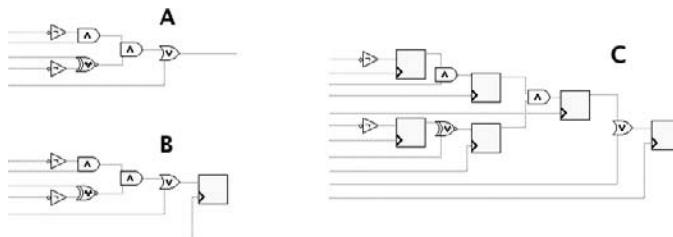
## Synchronous execution and the enable chain

LabVIEW is a data-flow-driven programming language, and LabVIEW FPGA takes special care to enforce dataflow in the generated code. This is done by transparently adding synchronization logic to code



**Figure 16.5** This FPGA project uses Emulation mode. All LabVIEW's debugging tools are available for use on the FPGA in Emulation mode. You can use the actual I/O hardware of the FPGA target as I/O in emulation, which allows you to verify algorithm functionality.

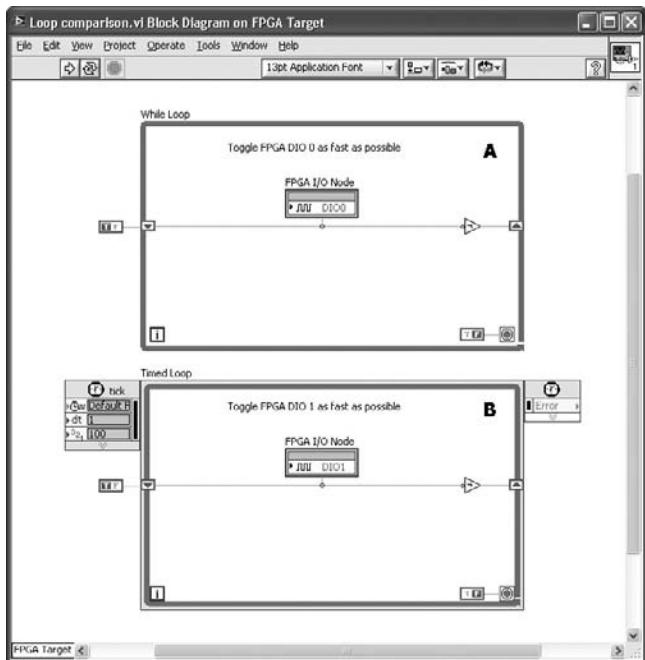
on your block diagram. Figure 16.6A shows a few simple logic functions chained together. What you might expect, if you are familiar with VHDL programming, is for all the logic functions to map to combinatorial logic and execute in a single clock cycle (as shown in Figure 16.6B). But LabVIEW treats each function as a separate entity requiring synchronization at its inputs and outputs, and LabVIEW does not optimize the code across synchronization boundaries. Instead of getting the code in Figure 16.6B, you get code that functions like that in Figure 16.6C. This is a huge difference in performance; if each function takes one clock tick, we've gone from one tick to four ticks of the clock to execute the same code. Figure 16.6C doesn't tell the whole story though,



**Figure 16.6** (A) What you drew. (B) What you wanted. (C) What you got. LabVIEW enforces dataflow on the VHDL code it generates.

because transparently added to each function is an extra input called the **enable chain**. The combination of synchronous execution and the enable chain enforces dataflow but also uses FPGA real estate and slows down execution.

A good way to illustrate the effect of the synchronization logic and enable chain on execution is to see how fast you can toggle a digital line on a RIO card. There are two loops in Figure 16.7. Loop A is a normal While Loop, and loop B is a single-cycle Timed Loop. Each loop runs as fast as possible and toggles a digital line. We can benchmark loop performance by monitoring the digital lines on an oscilloscope. With the VI compiled to use a 40-MHz clock, you would expect to see a 20-MHz square wave output from both loops. But the dataflow nature means each piece of the block diagram adds overhead and loop A takes three ticks to execute, resulting in a 6.67-MHz square wave. Loop B, however, generates a 20-MHz square wave because it is a single-cycle Timed Loop. A single-cycle Timed Loop uses clocked execution and executes all the code inside the loop within a single clock cycle. You can maximize performance and avoid the enable chain with a single-cycle loop.

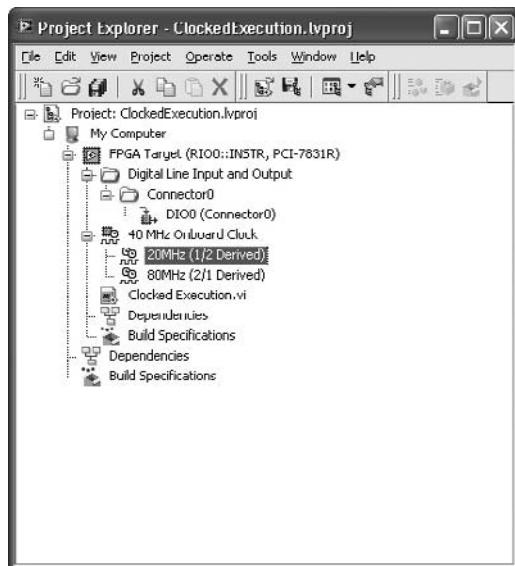


**Figure 16.7** Two loops toggle digital lines as fast as possible. (A) Loop overhead and the enable chain cause loop A to execute in 3 clock ticks. (B) Single-cycle Timed Loops execute completely with each clock tick.

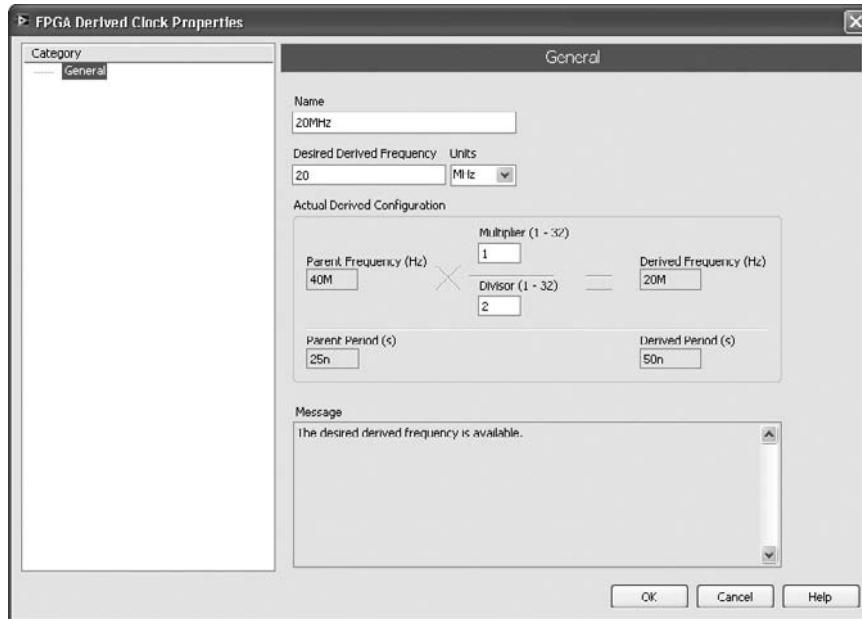
## Clocked execution and the single-cycle Timed Loop

Clocked execution was introduced in LabVIEW 7.1 with the single-cycle Timed Loop. Code placed within a single-cycle Timed Loop is specially optimized to execute on a single clock tick. Logic functions in the loop map directly to combinatorial FPGA logic. LabVIEW shift registers map directly to FPGA flip-flops. The flip-flops hold state from one clock cycle to the next. This leads to an efficient FPGA implementation, but unfortunately not all LabVIEW functions are able to map directly to logic or are capable of executing in a single clock tick. Only combinations of logic capable of executing in a single clock tick can be placed in a single-cycle loop. Look in the online help for a list of unsupported functions for the single-cycle Timed Loop.

One workaround you can use to fit more logic into a single clock tick is to slow down the clock. In the project in Figure 16.8 we have our 40-MHz clock and two derived clocks, one at 20 MHz and the other at 80 MHz. You can have multiple clocks in an FPGA project using the onboard clock to create derived clocks. To create a new derived clock, right-click on the 40-MHz onboard clock in the project window and select New FPGA Derived Clock. Figure 16.9 shows how to configure your new clock. Once it is configured, you can use it in your project. The 7831R target we are using normally compiles with a 40-MHz clock, but

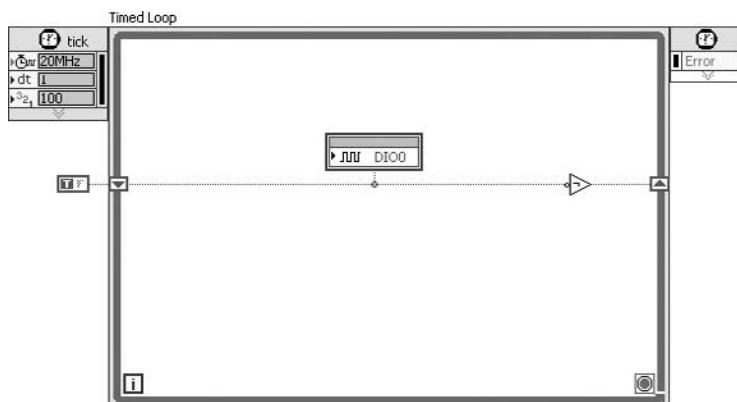


**Figure 16.8** You can have multiple clocks in an FPGA project. On this target they are all derived from the 40-MHz onboard clock.

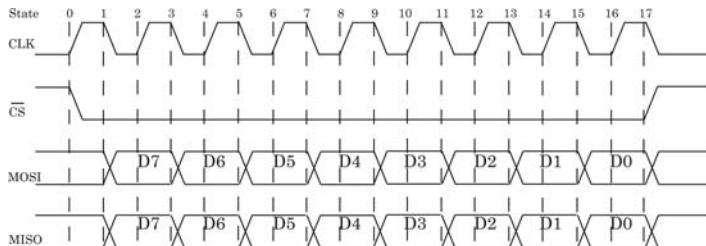


**Figure 16.9** New FPGA Derived Clock dialog. Configure new derivative clocks based on the onboard clock for your target.

we could change the top-level clock to use our 80-MHz derived clock if we needed things to run a little faster. Using a faster top-level clock allows the code to execute faster, but it also introduces jitter into the system. If you set a clock rate that is too fast, the VI will not compile and you will have to select a lower clock and compile again. Figure 16.10



**Figure 16.10** Single-cycle Timed Loops can use derived clocks as their timing source.



**Figure 16.11** Timing diagram for an SPI communication. Counting clock edges provides an easy way to determine states.

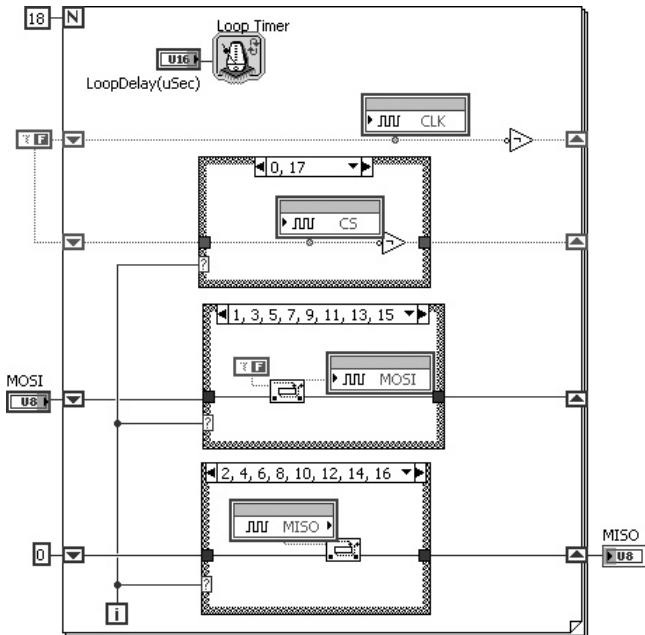
shows a single-cycle Timed Loop with a 20-MHz derived clock as its time base. Clocked execution with the single-cycle Timed Loop and derived clocks provides a lot of flexibility.

## Parallelism

Programming for an FPGA is distinctively different from that for LabVIEW on a desktop. An FPGA executes parallel instructions in parallel whereas a microprocessor executes instructions sequentially. It's important to take advantage of this parallelism when you develop your program. A parallel state machine turns out to be an easy way to build a serial bus controller. Figure 16.11 shows a timing diagram for a simple serial peripheral interface (SPI) communication sequence. SPI is a common bus in chip-to-chip communications between microprocessors and peripheral components. One of the strengths of LabVIEW FPGA is the ability to interface with low-level devices using communications busses such as SPI. This lets you build test systems that interface directly with board-level components. It is fairly easy to design an SPI bus controller by counting clock edges on a timing diagram. Each component will have a diagram in its data sheet, and it's fairly simple to get out a ruler and mark out the transitions. Each clock edge represents a different state. The Chip Select line is set low in state 0 and high in 17. Data is latched out on the MOSI (master out slave in) line on each falling edge. The MISO (master in slave out) latches in data on each rising edge. Figure 16.12 illustrates how parallel execution with a separate case for each line results in a clean, easy-to-understand diagram.

## Pipelining

A powerful extension of parallelism is pipelining. Pipelining breaks a series of sequential tasks into parallel steps for faster loop rates. This parallel execution can end up increasing the throughput of a sequential task. The total time for the code to execute is still the sum of the time required for each individual task, but the loop rate is faster because

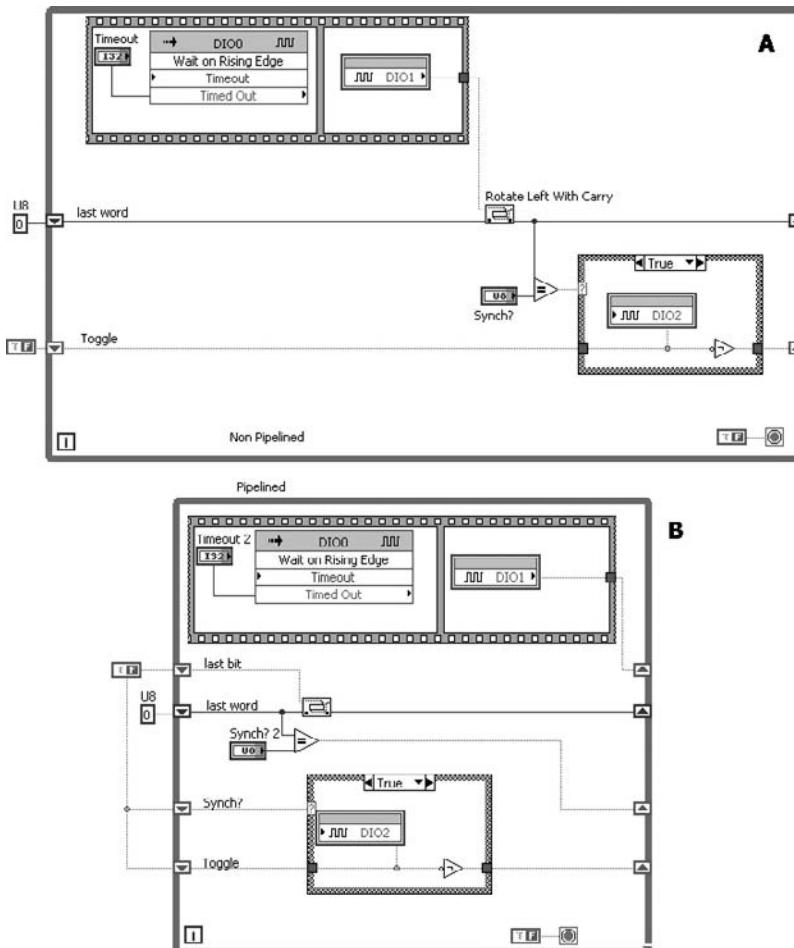


**Figure 16.12** SPI controller as a parallel state machine. CLK toggles each iteration. CS is asserted low at 0 and high at 17. MOSI clocks out data 1 bit at a time at states 1, 3, 5, 7, 9, 11, 13, 15. MISO reads in data at 2, 4, 6, 8, 10, 12, 14, 16. The other cases are empty.

each piece of code executes in parallel. The diagram in Figure 16.13 show two identical tasks performing the same function. The sequence frame waits for a rising edge on DIO0 before reading the bit on DIO1. Data is shifted in and compared for a match with a synch word. When a match is found, DIO2 is toggled. Figure 16.13A shows the nonpipelined approach. Each action happens sequentially. Figure 16.13B shows the pipelined version. There are still three distinct sections of code, but now they happen in parallel. Pipelining this process does not reduce the total time between the rising edge on DIO0 and the toggle of DIO2, but it does allow the VI to work with a faster clock on DIO0.

## Conclusions

The great thing about programming with LabVIEW FPGA is that it's just LabVIEW. You don't have to learn another language or development environment. The hardware interface tool LabVIEW FPGA puts in your toolbox is powerful. Once you understand the subtleties of the hardware target, you'll wonder how you ever did without it. Just remember to keep your code clean and simple, take advantage of the



**Figure 16.13** Pipelined versus nonpipelined approaches to a problem. The pipelined version is more efficient and can work with a faster clock on DIO0.

inherent parallelism in the FPGA, and verify through each step of development. Also be aware of the impact of I/O arbitration on loop speed. By default, LabVIEW places extra flip-flops as arbiters, but they add execution overhead. Incremental development and test is the best way to verify the impact of each function on your code.

## Bibliography

“Thinking Inside the Chip,” NIWeek04, National Instruments Corporation, 11500 N. Mopac Expressway, Austin, Tex., 2004.

“Virtex-II Platform FPGAs: Complete Data Sheet,” DS031 (v. 3.3), Xilinx, Inc., 2100 Logic Drive, San Jose, Calif., June 24, 2004.

*This page intentionally left blank*

# LabVIEW Embedded

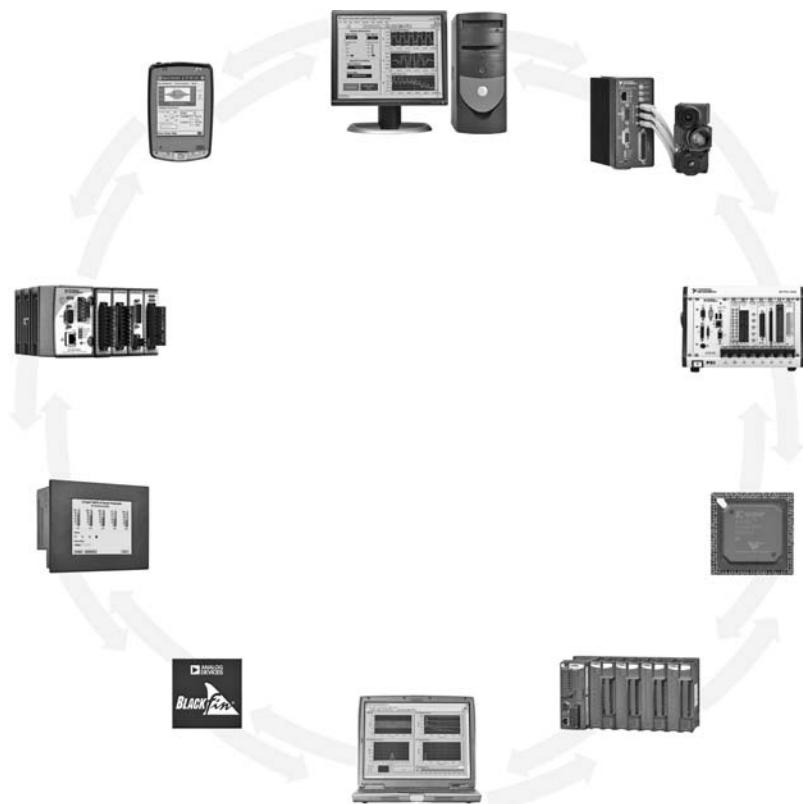
In the last edition of this book we showed you how to build an embedded system using LabVIEW for Linux and a PC104 computer. It was big and power-hungry, but at the time it was the only way to build an embedded system with LabVIEW. National Instruments has been hard at work on its own version of LabVIEW for embedded systems, and it's so much better than anything you or I could do. LabVIEW can now target any 32-bit microprocessor (Figure 17.1) To get an inside look at LabVIEW Embedded, we asked P. J. Tanzillo, National Instruments' product support engineer for the LabVIEW Embedded platform, to give us a hand and tell us what this new technology is all about. So, without further ado, here's P. J.:

## Introduction

LabVIEW is much more than a tool for instrument control and data acquisition. With its extensive built in signal processing and analysis libraries, LabVIEW has developed into an ideal environment for algorithm development and system design. Furthermore, with the addition of the LabVIEW modules like the LabVIEW **Real-Time** Module and the LabVIEW **FPGA** Module, it has become possible to run these applications not just on a Windows PC, but on other hardware platforms as well. In this chapter, we will examine the newest such module called the **LabVIEW Embedded Development Module**. This latest addition to the LabVIEW family allows you to develop applications for any 32-bit microprocessor.

## History

In the late 1990s, it became clear that Personal Digital Assistants (PDAs) were gaining mainstream acceptance as a viable computing platform. National Instruments saw this platform as a potential compliment to their



**Figure 17.1** LabVIEW targets devices from the desktop to the palmtop, and beyond. LabVIEW Embedded can target any 32-bit microprocessor. (*Photo courtesy of National Instruments*)

industrial control hardware, and so investigation began into how someone might use LabVIEW to program PDAs. The initial concept proposal included a potential use case: A factory foreman monitors and controls his/her headless automation controller with a simple LabVIEW Graphical User Interface on a handheld device.

At the time, however, PDAs were running on several processor architectures with various operating systems, and no single technology was emerging as a standard. Since LabVIEW is a compiled language, it would have been necessary to port the entire LabVIEW compiler and run-time engine in order to support a new architecture. Porting LabVIEW to a single new architecture would have been a significant effort, and this approach would have required several such efforts to address any significant number of PDA users.

Furthermore, the application memory space on PDAs was significantly smaller than any other platform that LabVIEW then supported.

Therefore, even more effort would have to be made to reduce the code footprint of LabVIEW built executables. It became clear that a new approach would be needed to port LabVIEW to these small, inconsistent targets.

Therefore, investigation began into creating a **LabVIEW C code generator**. This approach was chosen because if LabVIEW were to generate ANSI C, then it could use existing C cross-compilers to build the application for any PDA. Furthermore, the code footprint would not be an issue since LabVIEW could then take advantage of the aggressive code **dead-stripping** of C compilers.

After a significant development effort, this technology was released for the first time in 2003 as the LabVIEW PDA Module. As development continued and the technology matured, it became evident that this approach of C Code generation could be applied more generally toward embedded microprocessors. Since C cross-compilers already existed for almost any embedded platform, it was simply an issue of decoupling the LabVIEW C Code from the PDA and adding optimizations so that LabVIEW generates efficient C Code. Once that effort was completed, and once a framework and documentation was developed to help a user port LabVIEW to any target, the LabVIEW Embedded Development Module was released in May 2005.

### LabVIEW Embedded Development Module

From a high level, the goals of the LabVIEW Embedded Development Module were simple. National Instruments wanted to make programming of embedded systems easier by allowing the LabVIEW programming experience to be extended to arbitrary hardware. In many cases, a system's specifications can be met by a commercial off the shelf hardware platform like NI CompactRIO. Thus, the LabVIEW Real-Time Module is sufficient for such situations. However, there are many other cases where constraints such as size, power consumption, and production volume will make it necessary for custom hardware to be created. With the introduction of the LabVIEW Embedded technology, this second case is now achievable with LabVIEW as well.

Unfortunately, since the embedded development market is so fragmented with seemingly infinite combinations of processor architecture, I/O peripherals, and operating systems, it again proved impossible to identify the “correct” platform to meet the needs of any significant number of embedded programmers. Therefore, it became necessary to allow for LabVIEW generated C Code to be integrated with any operating system and embedded toolchain (Figure 17.2).

### The technology: What's happening

In order to get from a LabVIEW VI to a cross-compiled embedded application, several steps must be completed.

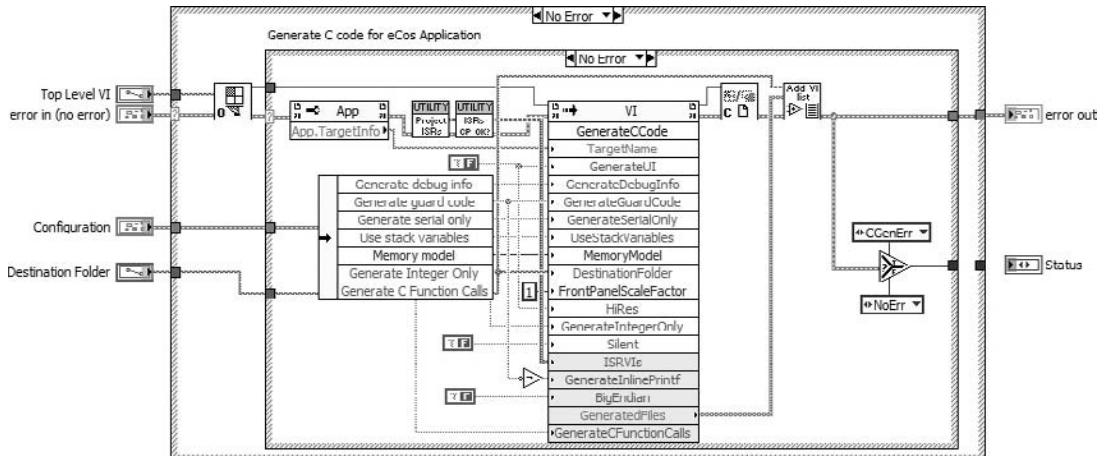


Figure 17.2 LabVIEW generates C code.

First, LabVIEW must traverse the block diagram and generate ANSI C code for the top-level VI as well as for every subVI in the VI hierarchy. The **LabVIEW C Code Generator** generates simple C primitives wherever possible. For example, while loops convert to `while()` statements, and the add function generates a simple '+'. More complex functions, however, cannot provide such a direct mapping. Therefore, an additional component of the LabVIEW generated C Code is the LabVIEW C Runtime Library. This is analogous to the LabVIEW run-time engine on the desktop, and we will discuss the LabVIEW runtime library in greater detail later in this chapter.

Next, all of the generated code must be passed through a **cross-compiler** that is specific to the desired target. In addition to the generated code, you can also add preexisting C source code to your LabVIEW Embedded Project. These files are passed through the cross-compiler with the generated code and **linked** into the **executable**. Once the compiling and linking is complete, you will have a cross-compiled executable that is specific to your target in your build folder along with all generated source and object files. This executable needs to be loaded into the correct location in memory on the device, and the appropriate execute command needs to be sent so that the application begins running. The typical communication mechanisms for downloading code and communicating with an embedded device are an RS-232 serial connection, an Ethernet connection, or a JTAG emulator.

Finally, in order to have live front panel controls and indicators, the target and the host need to establish communications. This can again be done through any protocol that can be understood by both the host and the target. This can be implemented in one of two possible ways.

The simpler method of the two is called **instrumented debugging**. In this approach, additional function calls are added to the generated C Code that handle sending updates back to LabVIEW. Since details of the communication protocol are built into the implementation of instrumented debugging, you only need to provide LabVIEW with implementations for open, read, write, and close. Common communications used for instrumented debugging are serial and TCP/IP, but other successful implementations have been seen using shared memory or CAN. The benefits of this approach lie mostly in the ease of implementation. In addition, there is no need for external JTAG emulators which can be very expensive and cumbersome. This approach, however, will expand the code footprint by as much as 40%, and since the additional functions are added to the generated code, there is a non-negligible overhead that will be incurred by the communication routines.

The other debug method is called **on-chip debugging**. This method does not call for any additional functions to be added to the generated code, so debug and release builds are identical. Rather than adding additional function calls, here the LabVIEW C code generator adds only additional comments. Each wire, control, and indicator on the block diagram results in a separate generated comment. At compile time, these comments are mapped to physical memory locations on the chip, and this information is stored in a **debug database**. When LabVIEW needs a value of a specific wire, indicator, or probe, it looks to the debug database to see what segment of physical memory that it needs to read. Then, it requests the value from the JTAG emulator/debugger using that debugger's provided API. This approach is considerably more complex, and thus, it requires more implementation effort. For example, the means of mapping lines of C code to physical memory locations is compiler dependent and non-trivial. In addition, if the debugging interface requires lengthy processor halts to read memory, this method can be rather intrusive to the program's execution. However, high quality probes that can efficiently read and write memory can result in almost completely non-intrusive debugging. In addition, full debugging information can be included without increasing the final code footprint.

### Running LabVIEW Embedded on a new target

Before you begin porting LabVIEW to your target, you must first be able to build, download, and run a simple application in C. This expertise is necessary to ensure that the **Board Support Package (BSP)** is correctly installed and configured. In addition, this will ensure that you have the necessary **toolchain** on the host so that you can compile and link the LabVIEW generated C code to create embedded applications. In addition, you will need to select hardware that is compatible. The following are some required and recommended specifications for a LabVIEW Embedded target.

**Hardware requirements:**

- 32-bit processor architecture
- 256K of application memory (plus whatever is required for your embedded OS)

**Hardware recommendations (but not requirements):**

- Operating System—This will allow you to take advantage of parallelism and multithreading in LabVIEW Embedded.
- Hardware Floating Point Unit—Since the LabVIEW Analysis Library is built on floating point math, there will be a significant performance gain if you are able to complete that math in hardware rather than software emulation.

Once your hardware, toolchain, and BSP are in place, you can begin the process of porting LabVIEW to your target. This consists of four main steps, each of which we will discuss in detail. They are:

Porting the LabVIEW Runtime Library

Incorporating the C Toolchain

Incorporating I/O Drivers

Customizing the LabVIEW Environment

The rest of this chapter will be spent discussing these four steps in detail.

**Porting the LabVIEW runtime library**

The LabVIEW C code generator will produce the same C code regardless of the target platform. Each LabVIEW function that does not map directly to a C primitive is implemented using a function call. These generic functions are implemented in the **LabVIEW C Runtime Library**. These functions are themselves implemented atop an even lower layer of generic functions that handle memory movement, string manipulation, simple timing, as well as a few other miscellaneous functions.

One of the major components of porting the LabVIEW C Runtime Library is providing a mapping of these generic low level function calls to functions that are specifically supported by your runtime environment. This will ensure that the LabVIEW generated C Code can be compiled for your target platform. This mapping needs to be completed for about 50 specific functions in one header file called **LVDefs\_plat.h**. Here, a series of #define statements map the generic function calls like `StrCaselessCompare( x, y )` to Operating System (OS) specific function calls like `strcasecmp( x, y )` (for Linux and VxWorks) or `_stricmp( x, y )` (for Windows). In addition to providing this mapping, `LVDefs_plat.h` also contains some important feature support flags such as defining whether or not a file system is present or whether TCP/IP or UDP communication is supported. Since most operating systems have migrated to similar APIs, there is no reason to begin

creating LVDefs\_plat.h from scratch. In most cases, it is wise to simply copy an existing implementation that is for an OS most like yours, and then begin to modify it where necessary.

In order for your target to support some of the more advanced features in LabVIEW, you will also need to implement some OS specific functions. For instance, in order for the timing primitives to be supported, you will need to implement the LVGetTicks() function. Similar functions are defined in the porting guide that are necessary for further LabVIEW synchronization and timing features like Timed Loops and Notifiers. Source files for the OS specific pieces of the LabVIEW C runtime library can be found in an operating system specific folder in the C Generation codebase.

Finally, the main entry point for all applications that LabVIEW generates can be found in LVEmbeddedMain.c. This file contains functions that perform set up and tear down of common pieces such as occurrences and FIFOs as well as any hardware specific setup and tear down routines that are necessary for a given target. The main function in LVEmbeddedMain.c initializes all global variables and then calls the top-level VI in the project. After the top-level VI has completed, a shutdown routine is completed. These initialization and shutdown routines are defined by two macros in LVEmbeddedMain.c called LV\_PLATFORM\_INIT and LV\_PLATFORM\_FINI. These macros are defined per target rather than per OS because different routines may be required for separate hardware platforms that share the same OS. For instance, although they both run eCos, a PowerPC for engine control and an ARM processor in the Nintendo Gameboy Advance would require very different hardware initialization routines.

### Incorporating the C toolchain

Once the LabVIEW Runtime Library has been ported, you should be able to manually compile and run LabVIEW generated C Code on your target. The next section will discuss the architecture of LabVIEW Embedded, the infrastructure that exists for incorporating your toolchain so that building, downloading, and debugging is an automated process.

Let's first review the process of cross-compiling a simple C program from a command line. It is important to understand this well, as this is the process that LabVIEW will need to invoke for a complete and ported target. A typical compiler command will consist of the following components:

```
compiler -XXX(C Flags) sourcefile.c -I Include files
```

The output of such a command will be an **object file** (\*.o) that will have the same file name as the C source file. A similar compile command will need to be executed for every C source file in your project (as well as the LabVIEW C runtime library). Once the compile step is completed, you will need to **link** all the compiled object files and pre-existing libraries into a single stand-alone **executable** file. This is done by running the compiler

command again, this time with appropriate flags set so that linking is performed. This command will typically look like this:

```
compiler -XXX(L Flags) "destinationfile" object files
```

Once the application has been successfully linked, you will have a cross compiled executable for your target platform. This is what we will be referring to as “building” your application. Though the syntax and order of components of the compiler for your target may vary, these key components will need to be in place for the compiler and the linker to succeed.

### The Embedded Project Manager

In order to generate C Code and build that generated code into an embedded application, you will need a LabVIEW Embedded Project (\*.lep) in addition to the VIs in your hierarchy. The .lep file contains information about the C Code generation options as well as any external source files that you wish to build along with the generated code. The LabVIEW Embedded Project Manager (Figure 17.3) is the common user interface that allows you to modify all of your projects for all of your targets. It provides a single source for all project configuration data (such as source files

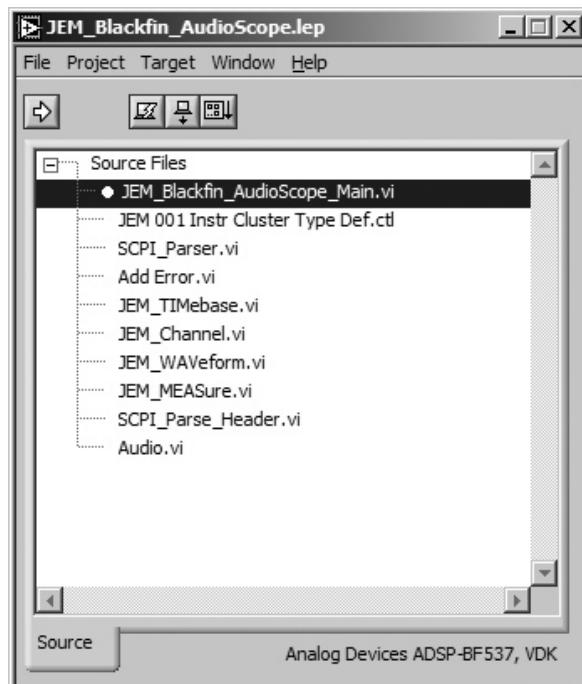


Figure 17.3 The Embedded Project Manager controls embedded builds.

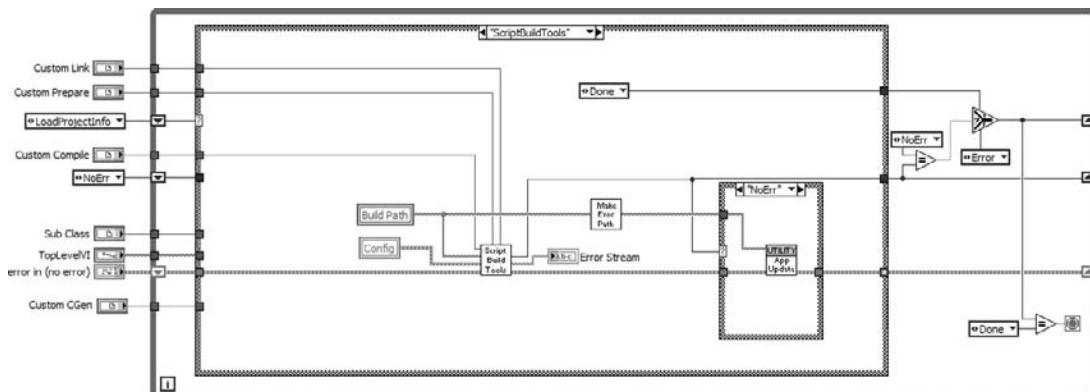
and additional compiler flags) as well as target specific settings (such as IP address or COM port settings).

The LabVIEW Embedded Project Manager is also the place where you interact with the target and the project. For example, there is a “Build” button that will cause LabVIEW to generate the C Code for all of the VIs in the project (as well as for the entire VI hierarchy for each of these VIs) and compile and link the generated code. The implementation of this and other actions within the LabVIEW Embedded Project Manager vary from target to target, and will therefore need to be modified when adding support for a new target.

## LEP plug-in VIs

Each LabVIEW Target has a series of implementation VIs called **Plug-In VIs**. These VIs govern the interaction between LabVIEW and the target specific toolchain. For example, when the user hits the “Build” button in the LabVIEW Embedded Project Manager, there is a plug-in VI executed that is called “XXX\_Build.vi” where XXX is a target specific prefix. This **Build VI** is responsible for pulling the build options and target information from the .lep file, translating these options into C code generation options and C compiler flags. Then, this VI executes all of the appropriate compiler commands such that at the end of its execution, a finished, cross-compiled executable is present in the project folder (Figure 17.4). Each target will have a different Build VI, and each will invoke a different compiler and linker. In order to keep a clear distinction between plug-in VIs for different targets, a naming convention and folder hierarchy has been introduced.

Once you have installed the LabVIEW Embedded Development Module, you will find a folder for each of the operating systems of the shipping example targets. Within each of these folders, you will find a folder called OS\_LEP\_TargetPlugin where OS is the name of the operating system.



**Figure 17.4** Build.vi turns your project into an executable. Only one case is shown for clarity.

Within the OS folders that support multiple hardware platforms, you will find an additional folder for each supported hardware platform. For instance in the vxworks folder, there are two additional folders called ixdp425 and cmd565. Each of these sub targets will have their own plug-in folder called OS\_HW\_LEP\_TargetPlugin where OS is the name of the operating system and HW is the name of the hardware platform. Each of the child targets will inherit much of their functionality from the parent while overriding only that which must be changed to support the specific hardware platform. Just as is the case with porting the LabVIEW C runtime library to your target, much of the code that exists in the plug-in VIs can be used generically for any target platform. Therefore, it is advisable to begin your implementation of a new target by copying an existing target and modifying it. An **Embedded Target Creation Utility** has been added to the LabVIEW Embedded Development Module to help automate the process of copying, renaming, and relinking an existing target. If none of the existing example targets meet the needs of your target platform, there is also a blank template target included with only the Code present that is common to all embedded targets.

### Target\_OnSelect

When exploring the target plug-in VIs for the first time, it makes the most sense to begin with the Target\_OnSelect.vi. This VI consists of a series of cases in a case structure, each of which corresponds to a target menu selection from the Embedded Project Manager. In the future, this VI will be replaced by a target specific xml file that will provide the mapping from the LabVIEW project to specific implementation VIs.

Common target menu selections like Build, Run, Debug, and Build Options will likely be present for most targets, but you have the option to add custom target menu options for your target by modifying the Target\_Menu.vi and the Target\_OnSelect.vi. For example, the LabVIEW Embedded Module for ADI Blackfin Processors includes an option to reset the Blackfin board. In the future, this information will also be defined in the target specific xml file.

### Other plug-in VIs

The Build plug-in VI is responsible for converting a VI to a cross compiled executable that can be run on the target platform. This task is primarily accomplished in the **ScriptBt.vi** (Bt is short for Build Tools), and it can be broken down into four primary steps:

#### Code generation

1. The **CGen** plug-in VI is responsible for traversing the block diagram of the Top-Level VI and generating ANSI C code for all of the applicable VIs in the project. This capability is built into LabVIEW Embedded, and

it is exposed through a VI server call. Once this VI is complete, the project folder should have a \*.c file for every VI in the VI hierarchy that is not part of the native LabVIEW function set. The input to this VI server call is a cluster of configuration options that will determine the specifics of the C code that is generated. For example, one of the elements of the configuration cluster is a Boolean control called **GenerateDebugInfo**. If this value is true, then additional function calls will be added to the generated code so that the embedded application will communicate back to LabVIEW in order to update front panel indicators and acquire values from front panel controls (this is called Instrumented Debugging). If the GenerateDebugInfo control is set to false (the default value), then this code will not be included in the generated C code. This option (and most of the other options that are discussed in this section) are exposed to the user through the Build Options dialog that can be found for all example targets in Target >> Build Options.... The following are some of the other most commonly used C generation options:

- **GenerateGuardCode.** Guard code prevents a user from making common mistakes that would typically cause an application to crash. For example, guard code can prevent dividing by zero and indexing out of range in an array.
- **GenerateSerialOnly.** The default generated C code has the same execution behavior as LabVIEW on the desktop. For example, parallel While Loops run in parallel. However, the LabVIEW C Code Generator generates code that uses cooperative multitasking in a single thread. Additional threads are used only by Timed Loops. The generated C code is difficult to read and does not resemble code that a human would write. Set this attribute to TRUE if you do not want to generate the cooperative multitasking code. The code is easier to read and usually runs much faster without the cooperative multitasking code. However, you lose the parallel execution behavior. Parallel While Loops execute in sequence with one While Loop executing only after the other While Loop completely finishes. This execution behavior most closely resembles subroutine priority VIs in LabVIEW for Windows.
- **DestinationFolder.** DestinationFolder indicates where you want the LabVIEW C Code Generator to save the generated C files. Unless the Incremental Build attribute is TRUE, any C files in the destination folder that have the same name as VIs in the build are overwritten.
- **GenerateIntegerOnly.** If a block diagram does not contain any floating-point data types, the GenerateIntegerOnly attribute generates C code that does not have any floating-point declarations or operations. You can link the generated code with a run-time library compiled with the \_Integer\_Only flag set to produce applications that run without hardware or software floating-point support. Generate-IntegerOnly does not support fixed-point operations in the Embedded Development Module.

- **GenerateCFunctionCalls.** Determines the calling interface to subVIs only if you set GenerateSerialOnly to TRUE. The calling interface to a subVI usually is generated in such a way that if an input to or output from the subVI is unwired, the default data is used. This increases the overall amount of code and generated data for a VI relative to what you might use in a normal C program. If all the inputs and outputs are wired, default data is not needed and a more efficient interface could be generated. Set this attribute to TRUE to generate the interface to all VIs as C style function calls without any default data initialization, which can reduce the code size by as much as 50% for a small VI. An error occurs if any input or output to any VI is unwired when the C code is generated.
  - **OCDIComments.** Extra comments are placed in the generated C code to help figure out where certain items such as wires and nodes, are in the generated code for use during non-intrusive debugging. These comments make the generated code harder to read. Use this option to turn off these comments when they are not needed. The LabVIEW Embedded Development Module ships with an example target called Code Generation Only. This target only generates C Code for a VI when the build button is pressed and does not attempt to compile the generated C Code. Furthermore, most of the C code generation options that have been discussed in this section are exposed as build options in the Target >> Build Options...dialog. Feel free to experiment with the build options using this target and examine the generated C Code to observe the differences.
2. The **Prepare.vi** is responsible for readying the project folder for the compilation processes as well as querying the \*.lep file and building up a list of source and include files based in the current project information. For the most part, the prepare VI is relatively consistent in implementation from target to target. In most cases, the Prepare.vi accepts the project configuration information (like the build settings and the project folder) as an input, and produces an array of source files, include files, and precompiled libraries to be used by the **ScriptCompiler** and **ScriptLinker** VIs as inputs. The LabVIEW C Code runtime library can either be pre-built and linked in as a library or the source files can be built each time the project is built. The advantage to re-building the C code runtime library each time is that the C Compiler can apply much more aggressive deadstripping to the object files and therefore make the code footprint of the final application considerably smaller. In order to achieve such an end, each of the files in the LabVIEW C runtime library should be added to the array of source files in the build VI. If code size is not a major issue, however, it is wise to pre-build the runtime library and add it to the array of libraries in the Prepare.vi. This will reduce the time that it takes to build a LabVIEW Embedded application considerably.
  3. The **ScriptCompiler.vi** is responsible for scripting the compiler for each of the files in the source file array that is produced by the Prepare.vi. This typically consists of a list of the generated C files (one for each VI in the VI hierarchy of the project) as well as any external C

files that the user has included in the project. In essence, this VI is responsible for building up the correct compiler command in a large string and executing this from the command line using the **SysExec.vi** from the LabVIEW functions palette. The inputs to this VI are typically the configuration options for the project, an array of source files, and an array of file paths that need to be included. The result of a successful execution of this VI is a project folder full of object files that will be used by the linker to produce the executable. To observe the build process of any target, you will only need to have the ScriptCompiler.vi open while you are building a project.

4. The **ScriptLinker.vi** works in much the same way as the ScriptCompiler.vi, only now, the command that is executed is the command necessary to link all of the object files to produce one stand alone executable. Just as you monitored the progress of the ScriptCompiler.vi in the previous section, you can also monitor the execution of the ScriptLinker.vi in a similar way.
5. The Debug.vi is responsible for initiating a connection between LabVIEW and the running application so that you can have an interactive LabVIEW debugging experience including probes, breakpoints, and live front panels. This can be completed in one of two ways—instrumented debugging (TCP, Serial, CAN, etc.) and on-chip debugging (JTAG). In either case, the Debug plug-in VI relies on the proper and complete implementation of the debugging plug-in VIs. Once one or both of these mechanisms are in place, you only need to begin the debugging session from the Debug.vi and LabVIEW will handle the background communications to the application. For instrumented debugging via TCP/IP four things need to happen:
  - The generated code has debug information included.
  - The compiler directive of UseTCPDebugging=1 is set.
  - The application is run with the correct host IP address as a parameter.
  - The niTargetStartTCPDebug is called after the application is running.

### Incorporating I/O drivers

In traditional embedded C programming, analog and digital I/O has typically been done only on a register level. Even C level driver APIs are a rarity, and most data acquisition has to be done by direct memory access. LabVIEW Embedded introduces a means of abstracting the low level implementation of such tasks in a unified API called **Elemental I/O** (Figure 17.5). Although the register level access still needs to be included in the implementation of the Elemental I/O nodes when porting to a new target it must only be done once per target platform. The result is a simple access point to the analog and digital peripherals that are available in your system. Furthermore, the Elemental I/O implementation changes based on the current target, making LabVIEW code that was developed on one platform completely portable to another supported platform.

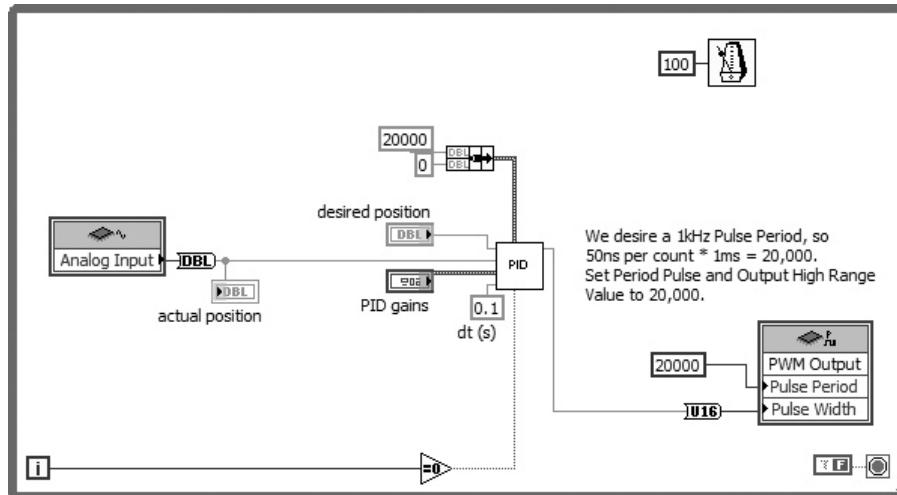


Figure 17.5 Elemental I/O provides simple access to analog and digital peripherals.

In order to enable easy register level programming in the LabVIEW environment, a new structure was added to LabVIEW Embedded called the **Inline C Node** (Figure 17.6). This new structure is used much in the same way that the Formula Node is used in core LabVIEW. However, the Inline C Node allows a user to add their own C and assembly code to the generated C code to facilitate easy register access and small code optimizations.

I/O driver development can differ greatly depending on the type and complexity of your device. For example, digital input and output is typically done by accessing the general purpose I/O pins on the device, and development of a driver can be relatively straightforward. However, a buffered DMA analog acquisition through an external A/D converter can be much more challenging to develop. In all cases, however, as with any I/O, there will typically be some initialization code that must be executed before the acquisition can be completed. In the traditional LabVIEW I/O programming model of Open -> Read/Write -> Close, the initialization code



Figure 17.6 Inline C node allows you to add your own C and assembly code directly on the LabVIEW block diagram.

obviously resides in the open VI. However, since an elemental I/O node consists of a single block, the implementation of this becomes slightly less apparent. To accomplish initialization within the Elemental I/O single node framework, it is best to declare a static integer in your Inline C node called “initialized.” This can act as a flag that should be set and reset accordingly as the acquisition is initialized and completed.

## LabVIEW Embedded programming best practices

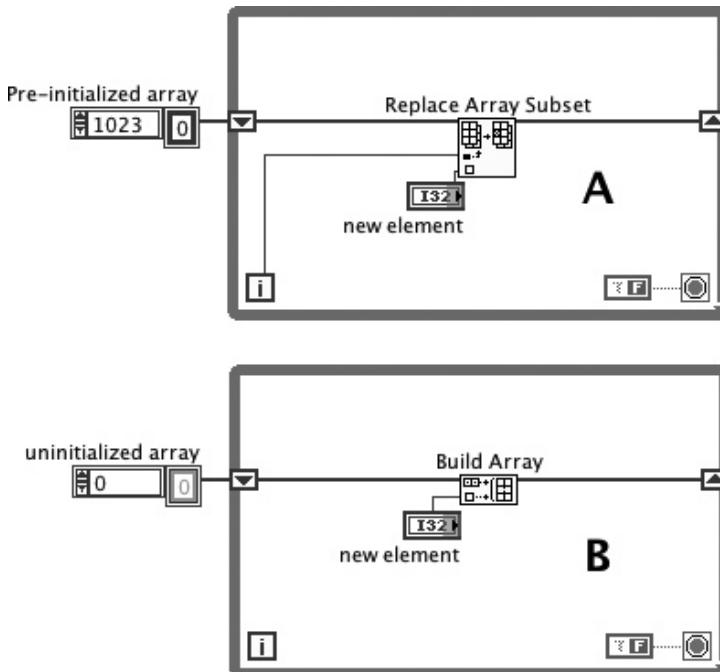
When developing an embedded application, system constraints such as memory limitations and time-critical code requirements can play a crucial role in your programming approach. This document outlines good programming practices that can be used to help optimize your embedded application with LabVIEW 7.1 Embedded Edition. The techniques described are applicable for LabVIEW on the desktop as well, but the benefit will be most evident on an embedded target.

**Avoid dynamic memory allocation.** Dynamic memory allocation is very expensive in time-critical code. In LabVIEW Embedded, dynamic memory allocation can occur when using the Build Array and Concatenate String functions. Alternatively, the Build Array primitive can be replaced with a Replace Array Subset function in order to replace elements in a pre-allocated array. The preallocated array should be created outside of the loop by using an array constant or with the Initialize Array function. LabVIEW code is shown below (Figure 17.7) to contrast the different implementations.

**Avoid placing large constants inside loops.** When a large constant is placed inside a loop, memory is allocated and the array is initialized at the beginning of each iteration of the loop. This can be an expensive operation in time-critical code. A better way to access the data is to place the array outside of the loop and wire it through a loop tunnel, or to use a global variable. Examples of the two recommended methods are shown below (Figure 17.8).

**Use global variables instead of local variables.** Every time a local variable is accessed, extra code is executed to synchronize it with the front panel. Code performance can be improved, in many cases, by using a global variable instead of a local. The global has no extra front panel synchronization code and so executes slightly faster than a local.

**Use shift registers instead of loop tunnels for large arrays.** When passing a large array through a loop tunnel, the original value must be copied into the array location at the beginning of each iteration, which can be expensive. The shift register does not perform this copy operation, but make sure to wire in the left shift register to the right if you don’t want the data values to change (Figure 17.8A).

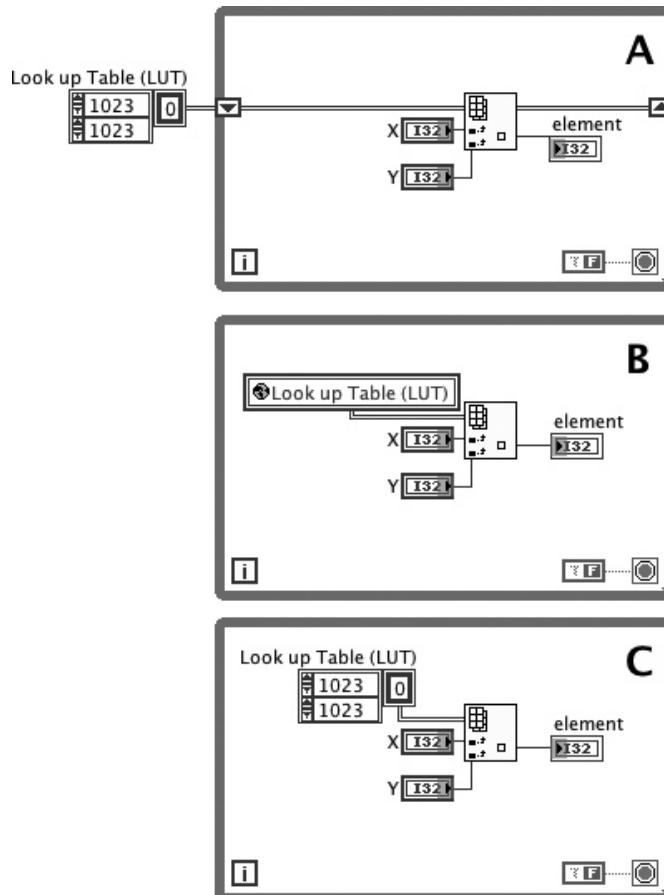


**Figure 17.7** Avoid dynamic memory allocation. (A) Array is pre-initialized and filled in inside the loop. (B) Build Array function allocates new memory each time it is called.

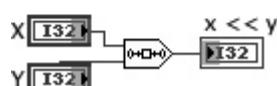
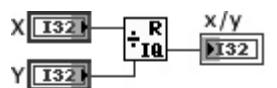
**Use integer operations instead of floating point operations.** If your processor does not have a floating point unit, converting to floating point to perform an operation and then converting back to an integer data type can be very expensive (Figure 17.9). In the examples below, using a Quotient & Remainder function is faster than a normal Divide function, and using a Logical Shift function is faster than a Scale by a Power of 2 function.

**Avoid automatic numeric conversions.** Another technique for improving code performance is to remove all implicit type conversions (coercion dots) (Figure 17.10). Use the Conversion functions to explicitly convert data types as this avoids a copy operation and a data type determination.

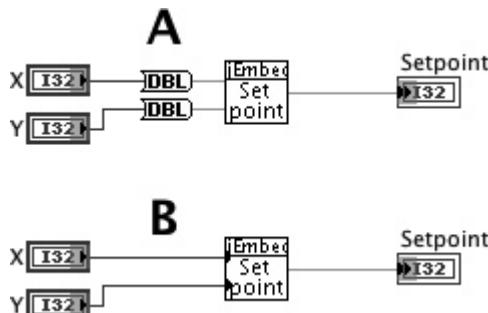
**Avoid Case structures for simple decision making.** For simple decision making in LabVIEW, it is often faster to use the Select function rather than a Case structure. Since each case in a Case structure can contain its own block diagram there is significantly more overhead associated with this structure when compared with a Select function. However, it is sometimes more optimal to use a case structure if one case executes a large amount of code and the other cases execute very little code. The decision to use a Select function versus a Case structure should be made on a case by case basis.



**Figure 17.8** (A) Memory for look up table is allocated once outside the loop. (B) A single copy of look up table is stored in global variable. (C) Memory for look up table is allocated each loop iteration.



**Figure 17.9** Integer operations are faster than floating point if your processor does not have a floating point unit.



**Figure 17.10** (A) Explicitly type convert inputs to subVIs. (B) Type coercion involves a data copy and data type determination.

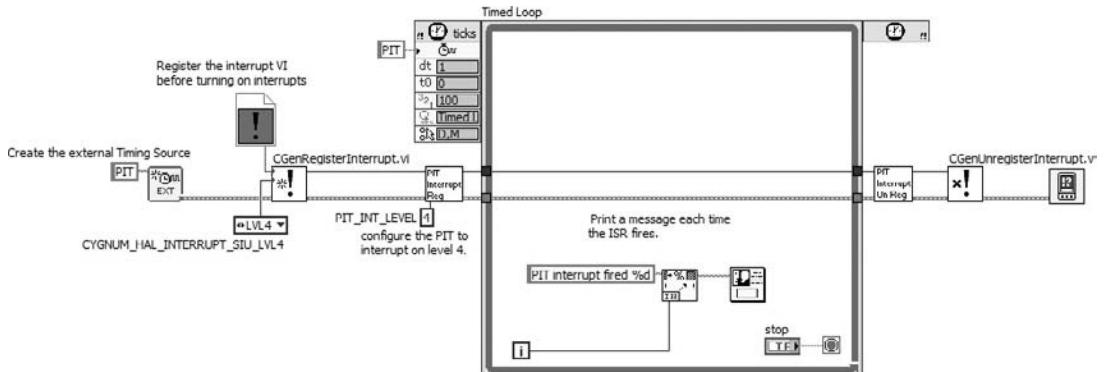
**Avoid In Range and Coerce in time-critical code.** The In Range and Coerce function has significant overhead associated with it due to the special user configurable features and extra data type determination operations. This function should be re-implemented with comparison and Select functions if it is used in time-critical code.

**Avoid clusters in time-critical code.** Clusters passed into subVIs will result in unnecessary data type information being passed as well. In order to speed up your code, try not to use clusters in time-critical areas of your block diagram.

Often the best results can be obtained by using a hybrid of LabVIEW and C code. The Inline C Node and **Call Library Node** allow the use of C code directly within your LabVIEW block diagram. See the Embedded Development Module documentation for more information on the use of the Inline C and Call Library Nodes. By following good embedded programming practices, you can better optimize your code to meet the constraints of your embedded application. Implementing one or two of these techniques may noticeably improve the performance of your application, but the best approach is to incorporate a combination of all these techniques.

### Interrupt driven programming

Many of you may be familiar with **event driven programming** on a PC. In this programming model, code is not executed sequentially. Rather, a user interacts with a user interface, and this interaction triggers some subset of code to be executed. If there is no user interaction, the program sits in an idle state waiting for the next event. The analogous programming model for embedded systems is called **interrupt driven programming**. In this model, hardware events are detected by the processor and designated code is executed in response; the specialized segments of code are called **Interrupt Service Routines (ISRs)**. These routines are typically



**Figure 17.11** Interrupt driven Timed Loop executes once per interrupt.

executed in a separate interrupt thread that is very high priority. Therefore, it is typical practice to spend as little time in the ISR as possible. Usually, such routines are only responsible for re-enabling the interrupt and setting some flag that can then be handled by the main program.

In LabVIEW Embedded, you can designate a VI as an ISR VI. Thus, you can develop G code that is triggered by some asynchronous hardware event, and this code will steal the execution from your top level VI at any point. However, since the ISR VIs do run in interrupt time, it is desirable to do as little in the ISR VI as possible.

The timed loop is a specialized version of the while loop that was added to LabVIEW 7.1. This structure allows you not only to control the loop rate, but also set independent priorities for the execution thread of that loop. In LabVIEW Embedded, this structure simply spawns an OS thread, and the embedded OS's scheduler handles scheduling the different threads. This is relevant to interrupt driven programming because you also have the ability to assign an external timing source to a timed loop. Furthermore, you can then programmatically fire a timed loop from an ISR VI. This allows for a block diagram with multiple parallel timed loops with each loop's timing source associated with some interrupt (Figure 17.11).

### LabVIEW Embedded targets

The LabVIEW Embedded Development Module comes with several example targets to serve as a model when porting to your custom hardware. For a complete list of supported hardware, visit [ni.com/embedded](http://ni.com/embedded).

*This page intentionally left blank*

## Process Control Applications

Industrial process control has its roots in the big process industries, sometimes called the Four P's: paper, petrochemicals, paint, and pharmaceuticals. These plants are characterized by having *thousands* of instruments measuring such variables as pressure, temperature, flow, and level, plus hundreds of control elements such as valves, pumps, and heaters. They use a great deal of automatic control, including feedback, sequencing, interlocking, and recipe-driven schemes. Modern control systems for these plants are, as you might imagine, very complex and very expensive. Most large process control systems are designed and installed through cooperative efforts between manufacturers, system integrators, and the customer's control engineering staff. These are the *Big Guns* of process control.

Chances are that you are probably faced with a smaller production system, laboratory-scale system, or pilot plant that needs to be monitored and controlled. Also, your system may need to be much more flexible if it's experimental in nature. Even though your needs are different, many of the concepts and control principles you will use are the same as those used in the largest plants, making it worth your while to study their techniques.

Large process control systems generally rely on networked minicomputers with a variety of smart I/O subsystems, all using proprietary software. Until recently, most software for these larger systems was not *open*, meaning that the end user could not add custom I/O interfaces or special software routines nor interconnect the system with other computers. Even the smaller process control packages—most of which run on PCs—have some of these lingering limitations. You, however, have an advantage—the power and flexibility of LabVIEW. It's not just a process control package. You can begin with a clean slate and few fundamental limitations.

To help LabVIEW users address the special needs of process control applications, National Instruments created an add-on toolkit that you install on top of LabVIEW. It's called the **Datalogging and Supervisory Control (DSC)** module. Take *everything* that we talk about in this chapter, do all the programming, add device servers (drivers) for popular industrial I/O, and you've basically got DSC. As always, if the capability of DSC doesn't meet your needs, you're still developing in LabVIEW, so you can do regular G programming and customize to suit. We liked DSC when it came out because it solved most of our process control challenges right out of the box. Whether you use DSC, or not, the rest of this chapter will make the whole picture clearer.

## Process Control Basics

In this chapter, we'll cover the important concepts of process control and then design the pieces of a small process control system, using good LabVIEW practices. First, a plug for the ISA: The U.S.-based Instrumentation, Systems, and Automation Society (**ISA**) sets many of the standards and practices for industrial process control. Gary joined the ISA some years ago while working on the control system for a large experimental facility, and he found his membership to be quite helpful. It offers a catalog of books, publications, standards documents, and practical training courses that can make your plant and your work practices safer and more efficient. The concepts and terminology presented in this chapter are straight out of the ISA publications, in an effort to keep us all speaking a common language. For information about membership or publications, contact the ISA at [www.isa.org](http://www.isa.org).

### Industrial standards

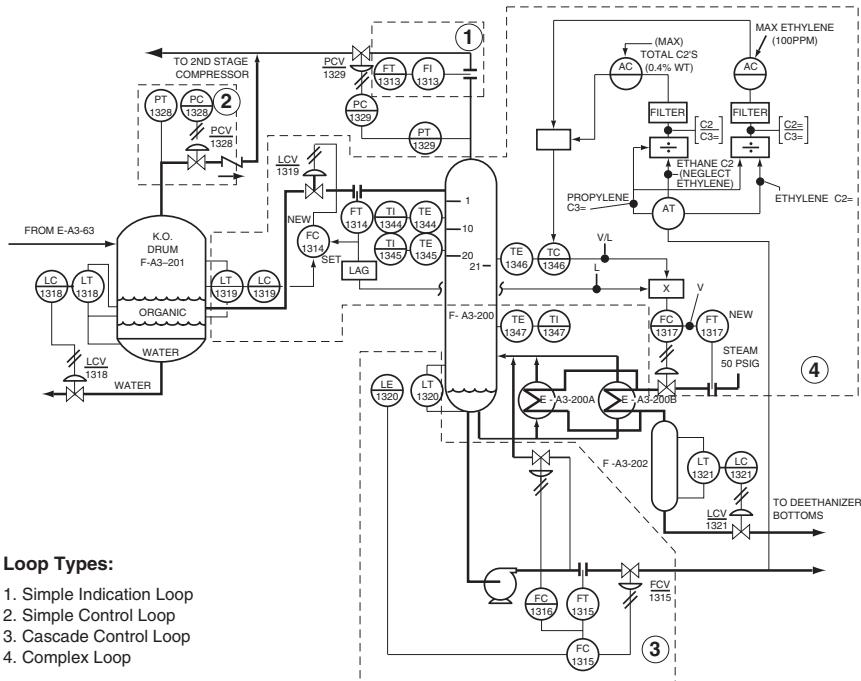
Standards set by the ISA and other organizations address both the physical plant—including instruments, tanks, valves, piping, wiring, and so forth—as well as the **man-machine interface (MMI)** and any associated software and documentation. The classic MMI was a silk-screened control panel filled with controllers and sequencers, digital and analog display devices, chart recorders, and plenty of knobs and switches. Nowadays, we can use software-based virtual instruments to mimic these classic MMI functions. And what better way is there to create virtual instruments than LabVIEW?

Engineers communicate primarily through drawings, a simple but often overlooked fact. In process control, drawing standards have been in place long enough that it is possible to design and build a large process plant with just a few basic types of drawings, all of which are thoroughly

specified by ISA standards. By the way, ISA standards are also registered as ANSI (American National Standards Institute) standards.

Some of these drawings are of particular interest to you, the control engineer. (See? We've already promoted you to a new position, and you're only on the third page. Now read on, or *you're fired*.)

**Piping and instrument diagrams and symbols.** The single most important drawing for your plant is the **piping and instrument diagram (P&ID)**. It shows the interconnections of all the vessels, pipes, valves, pumps, transducers, transmitters, and control loops. A simple P&ID is shown in Figure 18.1. From such a drawing, you should be able to understand all the fluid flows and the purpose of every major item in the plant. Furthermore, the identifiers, or **tag names**, of every instrument are shown and are consistent throughout all drawings and specifications for the plant. The P&ID is the key to a coherent plant design, and it's a place to start when you create graphical displays in LabVIEW.



**Figure 18.1** A piping and instrument diagram is the basis for a good process plant design.

**TABLE 18.1 Abbreviated List for the Generation of ISA Standard Instrument Tags**

	First letter measured or initiating variable	Second letter readout or output function	Succeeding letters (if required)
A	Analysis	Alarm	Alarm
B	Burner, combustion	User's choice	User's choice
C	Conductivity	Controller	Controller
D	Density/damper	Differential	
E	Voltage (elect)	Primary element	
F	Flow	Ratio/bias	Ratio/bias
G	Gauging (dimensional)	Glass (viewing device)	
H	Hand (manual)		High
I	Current (electrical)	Indicate	Indicate
J	Power	Scanner	
K	Time	Control station	
L	Level	Light	Low
M	Moisture/mass		Middle/intermediate
N	User's choice	User's choice	User's choice
O	User's choice	Orifice, restriction	
P	Pressure	Point (test) connection	
Q	Quantity	Totalize/quantity	
R	Radiation	Record	Record
S	Speed/frequency	Safety/switch	Switch
T	Temperature	Transmitter	Transmitter
U	Multipoint/variable	Multifunction	Multifunction
V	Vibration	Valve, damper, louver	Valve, damper, louver
W	Weight	Well	
X	Special	Special	Special
Y	Interlock or state	Relay/compute	Relay/compute
Z	Position, dimension	Damper or louver drive	

SOURCE: From S5.1, *Instrumentation Symbols and Identification*. Copyright 1984 by Instrument Society of America. Reprinted by permission.

Tag names should follow ISA standard S5.1, a summary of which is shown in Table 18.1. The beauty of this naming system is that it is both meaningful and concise. With just a few characters, you can determine what the instrument controls or measures (for example, *P* for pressure, *T* for temperature) as well as its function (for example, *C* for controller, *V* for valve, *T* for transmitter). A numeric suffix is appended as a hierarchical serial number. Most plants use a scheme where the first digit is the zone or area, the second is the major vessel, and so forth. Tag names appear everywhere—on the P&ID, instrument data sheets, loop diagram, in your control system database, and, of course, on a little metal tag riveted to the instrument. Since this is the only national standard of its type, most process engineers are familiar with this naming scheme—a good reason to consider its use.

When you write a LabVIEW program for process control, you should use tag names everywhere. On the diagram, you can label wires,

cluster elements, and frames of Case and Sequence structures that are dedicated to processing a certain channel. On the panel, tag names make a convenient, traceable, and unambiguous way to name various objects.

Here are some examples of common tag names with explanations:

PI-203 Pressure Indicator—a mechanical pressure gauge

LT-203 Level Transmitter—electronic instrument with 4–20-mA output

TIC-203 Temperature Indicating Controller—a temperature controller with readout

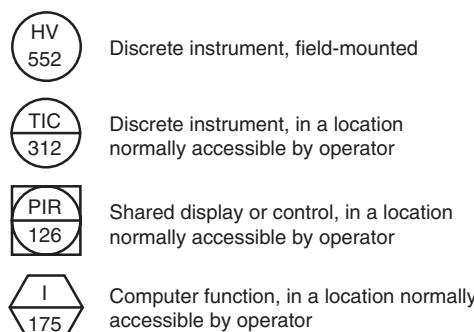
PSV-203 Pressure Safety Valve—a relief valve

FCV-203 Flow Control Valve—valve to adjust flow rate

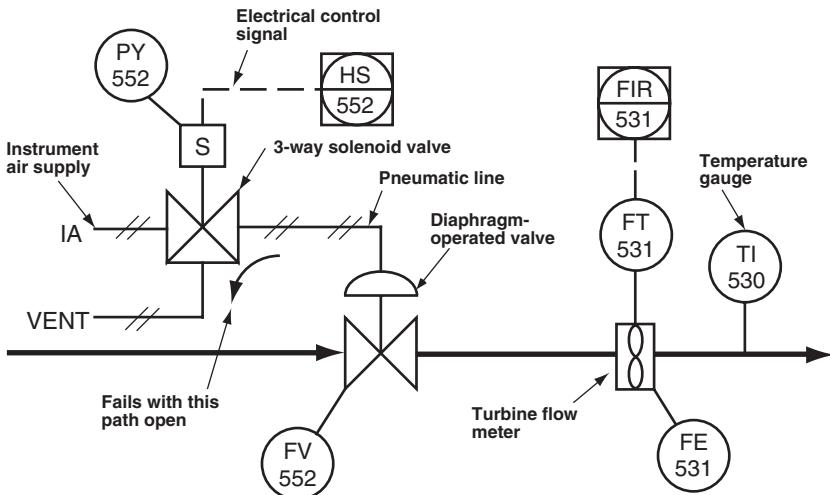
ZSL-203 Position Switch, Low-Level—switch that closes when a valve is closed

The little balloons all over the P&ID contain tag names for each instrument. Some balloons have lines through or boxes around them that convey information about the instrument's location and the method by which a readout may be obtained. Figure 18.2 shows some of the more common symbols. The intent is to differentiate between a field-mounted instrument, such as a valve or mechanical gauge, and various remotely mounted electronic or computer displays.

Every pipe, connection, valve, actuator, transducer, and function in the plant has an appropriate symbol, and these are also covered by ISA standard S5.1. Figure 18.3 shows some examples that you would be likely to see on the P&ID for any major industrial plant. Your system may use specialized instruments that are not explicitly covered by the standard. In that case, you are free to improvise while keeping with



**Figure 18.2** Function symbols, or balloons, used to identify instruments on the P&ID.



**Figure 18.3** Some instrument symbols, showing valves, transmitters, and associated connections. These are right out of the standards documents; your situation may require some improvising.

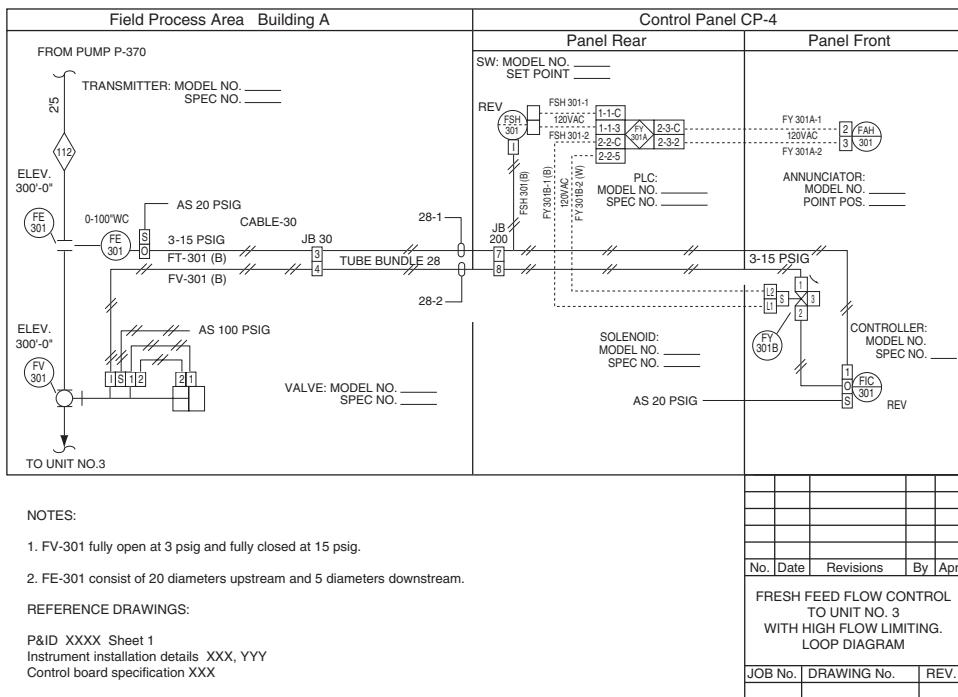
the spirit of the standard. None of this is *law*, you know; it's just there to help. You can also use replicas of these symbols on some of your LabVIEW screens to make it more understandable to the technicians who build and operate the facility.

**Other drawing and design standards.** Another of our favorite ISA standards, S5.4, addresses **instrument loop diagrams**, which are the control engineer's electrical wiring diagram. An example appears in Figure 18.4. The reasons for the format of a loop diagram become clear once you have worked in a large plant environment where a signal may pass through several junction boxes or terminal panels before finally arriving at the computer or controller input. When a field technician must install or troubleshoot such a system, having one (or only a few) channels per page in a consistent format is most appreciated.

Notice that the tag name appears prominently in the title strip, among other places. This is how the drawings are indexed, because the tag name is *the* universal identifier. The loop diagram also tells you where each item is located, which cables the signal runs through, instrument specifications, calibration values (electrical as well as engineering units), and computer database or controller setting information.

We've found this concise drawing format to be useful in many laboratory installations as well. It is easy to follow and easy to maintain.

There are commercial instrument database programs available that contain built-in forms, drawing tools, and cross-referencing capability



**Figure 18.4** An instrument loop diagram, the control engineer's guide to wiring. (Reprinted by permission. Copyright ©1991 by the Instrument Society of America. From S5.4, Instrument Loop Diagrams.)

that make management of large process control projects much easier. Ask ISA for a catalog of process control products, if you're interested.

Here are a few other ISA standards that can be valuable to you:

### S5.5, Graphic Symbols for Process Displays

#### S51.1, Process Instrumentation Terminology

#### S50.1, Compatibility of Analog Signals for Electronic Industrial Process Equipment

If you are involved with the design of a major facility, many other national standards will come into play, such as the *National Electrical Code* and the *Uniform Mechanical Code*. That's why plants are designed by multidisciplinary teams with many engineers and designers who are well versed in their respective areas of expertise. With these standards in hand and a few process control reference books, you might well move beyond the level of the mere LabVIEW hacker and into the realm of the registered professional control engineer.

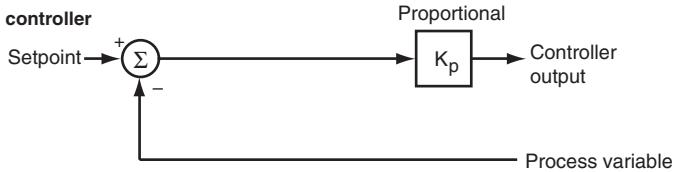
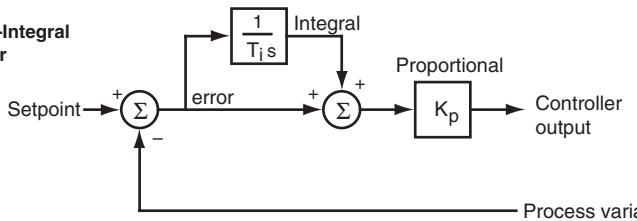
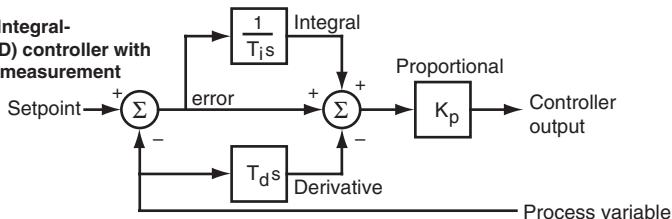
### Control = manipulating outputs

Controlling a process implies a need for some kind of output signal from the control system. (At last! Something that uses all those analog and digital output modules that somebody sold you!) The simplest control mode, **manual control**, relies on the operator to turn a knob or throw a switch to manipulate the process through some actuator. Programming a manual control system is about as simple as things get. The other control mode, **automatic control**, requires a hardware or software machine to make the adjustments. What both modes have in common (besides output hardware) is the use of **feedback**. In manual control, the operators see that the temperature is too high, and they turn down the heater. In automatic control, the controller makes a measurement (the **process variable**), compares it with the desired value, or **setpoint**, and adjusts the output, or **manipulated variable**, accordingly.

The output of the controller may be either digital or analog in nature. Digital-output controllers are also known as **on/off controllers** or **bang-bang controllers**. An example is the thermostat on your home's furnace. On/off controllers are generally the simplest and cheapest control technique, but they may be a bit imprecise—the process variable tends to cycle above and below the setpoint. Analog controllers, on the other hand, are proportional in nature, adjusting the output in infinitesimally small steps to minimize the error. They are powerful and versatile, but generally more complex and expensive than on/off controllers.

The most common types of automatic controllers in use today rely on the well-known **proportional-integral-derivative (PID)** family of algorithms. The simplest form of the algorithm is a pure **proportional (P)** controller (Figure 18.5A). An **error** signal is derived by subtracting the process variable from the setpoint. Error is then multiplied by a proportional gain factor, and the result drives the output, or **manipulated variable**. If the process reacts by “going in the right direction,” the process variable will tend toward the setpoint—the result of **negative feedback**. If the process goes the wrong direction, **positive feedback** results, followed by undesirable oscillation or saturation. By raising the proportional gain, the manipulated variable is more strongly driven, thus reducing the error more quickly. There is a practical limit to gain, however, due to delay (**lag**) in the process that will eventually cause a destabilizing phase shift in the control loop—the loop ends up responding to old information. Therefore, some residual error is always present with a proportional controller, but the error magnitude may be sufficiently small for your requirements that a simple proportional controller is adequate.

One way to eliminate this error is to mathematically **integrate** or **reset** the error over time. Combining these two control techniques

**A) Proportional controller****B) Proportional-Integral (PI) controller****C) Proportional-Integral-Derivative (PID) controller with derivative on measurement**

**Figure 18.5** Signal flow diagrams for proportional, integral, and derivative algorithms are the basis for much of today's practical feedback control. These are just a few examples of P/PI/PID configuration; there are many more in actual use.

results in the **proportional-integral (PI)** controller (Figure 18.5B). With the PI algorithm, the integral reduces the error by spreading the process reaction over time, thus giving it time to respond to controller commands. Like any control algorithm, too much of a good thing may result in substandard or unstable performance. Also, the integral can **wind up** (where the value of the integral grows to a very large value) when the process is sufficiently out of control for a long period of time or when the controlled device does not respond as expected. This can result in very long recovery times, so most controllers include an **anti-reset windup** feature to limit integral action. PI controllers are the most widely used in general applications.

When a process is subject to sudden upsets or large changes in set-point, the controller may not respond quickly enough. In such cases, a **derivative, or rate, term** may be added (Figure 18.5C). Taking the derivative of the error (a common technique) effectively increases the controller's gain during periods when the process variable is changing, forcing a quick correcting response. Unfortunately, the derivative is a kind of high-pass filter that emphasizes the controller's response to noise. Therefore, the full PID algorithm can only be used when the

signal has little noise or where suitable filtering or limiting has been applied.

National Instruments offers a set of PID algorithms available in the **PID Control Toolkit**. You can use them to build all kinds of control schemes; usage will be discussed later in this chapter. PID control can also be accomplished through the use of external “smart” controllers and modules. Greg Shinskey’s excellent book *Process Control Systems* (1988) discusses the application, design, and tuning of industrial controllers from a practical point of view. He’s our kinda guy.

There are many alternatives to the common PID algorithm so often used in industrial control. For instance, there are algorithms based on *state variable analysis* which rely on a fairly accurate model of the process to obtain an optimal control algorithm. *Adaptive* controllers, which may or may not be based on a PID algorithm, modify the actions of the controller in response to changes in the characteristics of the process. *Predictive* controllers attempt to predict the trajectory of a process to minimize overshoot in controller response. Modern *fuzzy logic* controllers are also available in LabVIEW in the PID Toolkit. That package includes a membership function editor, a rule-base editor, and an inference engine that you can incorporate into your system. If you have experience in control theory, there are few limitations to what you can accomplish in the graphical programming environment. We encourage you to develop advanced control VIs and make them available to the rest of us. You might even make some money!

So far, we have been looking at **continuous control** concepts that apply to steady-state processes where feedback is applicable. There are other situations. **Sequential control** applies where discrete, ordered events occur over a period of time. Valves that open in a certain order, parts pickers, robots, and conveyors are processes that are sequential in nature. **Batch processes** may be sequential at start-up and shutdown, but operate in steady state throughout the middle of an operation. The difficulty with batch operations is that the continuous control algorithms need to be modified or compromised in some way to handle the transient conditions during start-up, shutdown, and process upsets. A special form of batch process, called a **recipe operation**, uses some form of specification entry to determine the sequence of events and steady-state setpoints for each batch. Typical recipe processes are paint and fuel formulation (and making cookies!), where special blends of ingredients and processing conditions are required, depending on the final product.

Early sequential control systems used relay logic, where electromechanical switching devices were combined in such a way as to implement boolean logic circuits. Other elements such as timers and stepper switches were added to facilitate time-dependent operations. System

inputs were switch contacts (manual or machine-actuated), and outputs would drive a variety of power control devices such as contactors.

Most modern sequential control systems are based on **programmable logic controllers (PLCs)** which are specialized industrial computers with versatile and nearly bulletproof I/O interface hardware. Millions of PLCs are in service in all industries, worldwide, and for good reasons: They are compact, cost-effective, reliable, and easy to program. It's much easier to modify a PLC's program than it is to rip out dozens of hardwired relays mounted in a rack. PLCs can be networked, and LabVIEW turns out to be a fine man-machine interface.

### Process signals

The signals you will encounter in most process control situations are low-frequency or dc analog signals and digital on/off signals, both inputs and outputs. Table 18.2 lists some of the more common ones. In a laboratory situation, this list would be augmented with lots of special analytical instruments, making your control system heavy on data acquisition needs. Actually, most control systems end up that way because it takes lots of information to accurately control a process.

Industry likes to differentiate between **transducers** and **transmitters**. In process control jargon, the simple transducer (like a thermocouple) is called a **primary element**. The signal conditioner that connects to a primary element is called a *transmitter*.

In the United States, the most common analog transmitter and controller signals are 4–20-mA current loops, followed by a variety of voltage signals including 1–5, 0–5, and 0–10 V. Current loops are preferred because they are resistant to ground referencing problems and voltage drops. Most transmitters have a maximum bandwidth of a few hertz, and some offer an adjustable time constant which you can use to optimize high-frequency noise rejection. To interface 4–20-mA signals to an ordinary voltage-sensing input, you will generally add a 250- $\Omega$  precision resistor in parallel with the analog input. (If you're using National Instruments' signal conditioning—particularly SCXI—order its 250- $\Omega$  terminating resistor kits.) The resulting voltage is then 1–5 V. When

**TABLE 18.2 Typical Process Control Signals and Their Usage**

Analog inputs	Digital inputs	Analog outputs	Digital outputs
Pressure	Pressure switch	Valve positioner	Valve open/close
Temperature	Temperature switch	Motor speed	Motor on/off
Flow rate	Flow switch	Controller setpoint	Indicator lamps
Level	Level switch	Heater power	
Power or energy	Position switch		

you write your data acquisition program, remember to subtract out the 1-V offset before scaling to engineering units.

On/off signals are generally 24 V dc, 24 V ac, or 120 V ac. We prefer to use low-voltage signals because they are safer for personnel. In areas where potentially flammable dust or vapors may be present, the National Electrical Code requires you to eliminate sources of ignition. Low-voltage signals can help you meet these requirements as well.

The world of process control is going through a major change with emerging digital **field bus** standards. These busses are designed to replace the 4–20-mA analog signal connections that have historically connected the field device to the distributed control systems with a digital bus that interconnects several field devices. In addition to using multidrop digital communications networks, these busses use very intelligent field devices. They are designed for device interoperability, which means that any device can understand data supplied by any other device. Control applications are distributed across devices, each with function blocks that execute a given control algorithm. For instance, one control algorithm can orchestrate a pressure transmitter, a dedicated feedback controller, and a valve actuator in a field-based control loop.

The first of the digital communication protocols, **HART**, was created by Rosemount and supported by hundreds of manufacturers. It adds a high-frequency carrier (such as that used with 1200-baud modems) which rides on top of the usual 4–20-mA current loop signal. Up to 16 transmitters and/or controllers can reside on one HART bus, which is simply a twisted pair of wires. HART allows you to exchange messages with smart field devices, including data such as calibration information and the values of multiple outputs—things that are quite impossible with ordinary analog signals.

Perhaps the most important new standard is **Foundation Fieldbus** (from the Fieldbus Foundation, naturally), which has international industry backing. The ISA is also developing a new standard, **SP-50**, which will ultimately align with Foundation Fieldbus and other standards. Because the field devices on these field bus networks are so intelligent, and because the variables shared by devices are completely defined, the systems integration effort is greatly reduced. It is thus much more practical to implement a supervisory system with LabVIEW and one or more physical connections to the field bus network. You no longer need drivers for each specific field device. Instead, you can use drivers for *classes* of field devices, such as pressure transmitters or valves. National Instruments is an active player in the Foundation Fieldbus development process, offering several plug-in boards and NI-FBUS host software. At this writing, the standard is well established

and is in wide use. As development continues, look for extensive LabVIEW support for Foundation Fieldbus—it's in the National Instruments catalog.

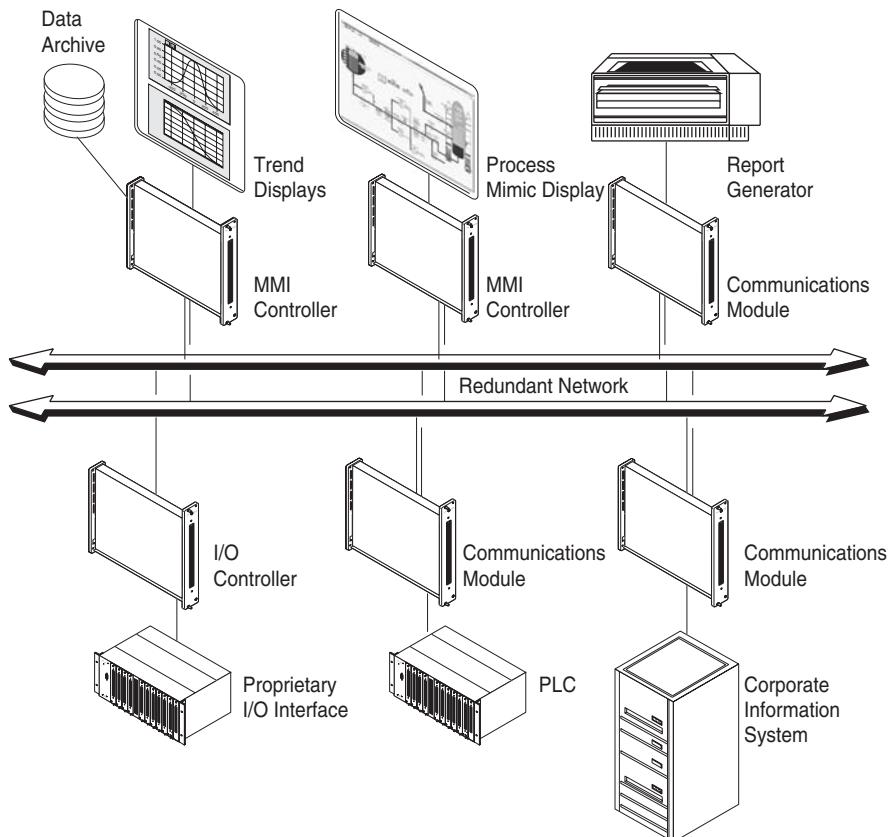
### Control system architectures

Hopefully, you will be designing your LabVIEW process control application along with the plant you wish to control. Choices of transducers and actuators, as well as the overall process topology, drastically affect the possibilities for adequate control. The process control engineer is responsible for evaluating the chemistry and physics of the plant with an eye toward controllability. The best software in the world can't overcome the actions of improperly specified valves, can't correct unusable data from the wrong type of flowmeter, nor can it control wildly unstable chemical reactions. You and your design team need to be on top of these issues during all phases of the project.

You will need to choose an appropriate control system architecture. There are many variables and many personal and corporate biases that point in different directions. Industrial control journals are filled with case histories and design methodologies that represent the cumulative knowledge of thousands of plant designs. For the novice, these journals are a source of inspiration, as are consultants and, of course, the major engineering firms. At the end of this section, we'll list some possible design parameters you might use to develop a checklist that leads to the right system.

**Distributed control system (DCS).** For the largest installations, distributed control systems have held the market for decades, though that position is being eroded by the improved capability of smaller, more open systems including PCs. Important characteristics of a DCS are as follows (see Figure 18.6):

- A hierarchical set of intelligent nodes, or controllers, ranging from smart I/O scanners and controllers to man-machine interface stations, on up to corporate-level management information systems
- Loop controllers physically close to the process being controlled
- Heavy reliance on local-area networks (usually redundant, for reliability)
- A globally shared, real-time database scheme where all nodes can freely share information
- Multiple levels of access security through passwords and communications restrictions



**Figure 18.6** A distributed control system, or DCS, encompasses many nodes communicating over networks and many I/O points.

- Integration of several different kinds of computers and I/O hardware
- Dozens, and sometimes hundreds, of nodes
- Many thousands of I/O points
- Very high cost, often in the millions of dollars

Older DCSs used proprietary operating systems and custom hardware, but new designs are moving more toward *open*, or public, standards in I/O hardware and operating systems such as Windows and Linux. All DCS vendors work closely with the customer because the system specification, configuration, and start-up phases are quite complex. *Turnkey* systems (where the system vendor does the lion's share of the work) are the rule rather than the exception.

A DCS may incorporate many kinds of computers and I/O subsystems, integrated into one (hopefully) harmonious package. In principle, a DCS could be designed with multiple systems running LabVIEW, but that may be stretching things a bit. At this time, it just isn't very well suited to this kind of massive application. Some major control system manufacturers are using LabVIEW at this time, but only in certain subsystems as part of an overall proprietary DCS. But given a few years, LabVIEW—particularly with DSC—may well penetrate these high-end applications.

**SCADA in the middle.** The middle ground in process control is handled by a broad range of systems collectively known as **Supervisory Control and Data Acquisition (SCADA)** systems (Boyer 1993). The name pretty well describes what they do. At the bottom of the architecture are the data acquisition features—measurements for the purpose of feedback and operator information. Next up the line are the control features, including manual commands and continuous and sequential automatic algorithms. At the top level are the supervisory aspects—the user interface, trending, report generation, and so forth. A key feature of SCADA is distributed I/O, including remote sensing and a heavy reliance on communications or networking with *remote terminal units*.

In contrast to a large DCS, SCADA systems are chosen for simpler systems where the total number of I/O points is unlikely to exceed a thousand, but with wide distribution. An example would be an oil or gas pipeline. You probably would not find control engineers at a large pulp plant who claim that their facility is run by a SCADA system. A small municipal waterworks is a more likely location. But the defining bounds of SCADA are becoming ever more fuzzy, thanks to the power and popularity of the personal computer and modern software.

**Enter the personal computer.** At the other end of the scale from a mega-DCS is the personal computer, which has made an incredible impact on plant automation. The DCS world has provided us with software and hardware integration techniques that have successfully migrated to desktop machines. A host of manufacturers now offer ready-to-run PC-based process control and SCADA packages with most of the features of their larger cousins, but with a much lower price tag and a level of complexity that's almost ... *human* in scale. LabVIEW with DSC fits into this category.

A wide range of control problems can be solved by these cost-effective small systems. In the simplest cases, all you need is a PXI system running LabVIEW with plug-in I/O boards or maybe some outboard I/O interface hardware. You can implement all the classical control schemes—continuous, sequential, batch, and recipe—with the functions

built into LabVIEW and have a good user interface on top of it all. Such a system is easy to maintain because there is only one programming language, and only rarely would you need the services of a consultant or systems house to complete your project. The information in this chapter can lead you to a realistic solution for these applications.

There are some limitations with any stand-alone PC-based process control system. Because one machine is responsible for servicing real-time control algorithms as well as the user interface with all its graphics, there can be problems with real-time response. If you need millisecond response, consider using outboard smart controllers (see the following section) or LabVIEW RT. The I/O point count is another factor to consider. Piling on 3000 analog channels is likely to bring your machine to its knees; you must consider some kind of distributed processing scheme.

LabVIEW, like the specialized process control packages, permits you to connect several PCs in a network, much like a DCS. You can configure your system in such a way that the real-time tasks are assumed by dedicated PCs that serve as I/O control processors, while other PCs serve as the man-machine interfaces, data recorders, and so forth. All the machines run LabVIEW and communicate via a local-area network using supported protocols such TCP/IP. The result is an expandable system with the distributed power of a DCS, but at a scale that you (and perhaps a small staff) can create and manage yourself. The problem is programming.

**PCs with smart controllers.** Another way to accommodate medium- and small-scale plants is to use your PCs with *smart* external controllers, such as PLCs and single-loop PID controllers, to off-load critical real-time tasks. This approach has some advantages. Dedicated, special-purpose controllers are famous for their reliability, more so than the general-purpose computer on which you run LabVIEW. They use embedded microprocessors with small, efficient executive programs (like miniature operating systems) that run your control programs at very high speeds with no interference from file systems, graphics, and other overhead.

PLCs can be programmed to perform almost any kind of continuous or sequential control, and larger models can be configured to handle thousands of I/O points and control loops, especially when networked. However, the PLC lacks a user interface and a file system, among other amenities—just the kinds of things that your LabVIEW program can provide. LabVIEW makes an ideal man-machine interface, data recorder, alarm annunciator, and so forth. It's a really synergistic match. About the only disadvantage of this approach is that you have two programming environments to worry about: LabVIEW plus the PLC's programming package.

**LabVIEW RT** enhances the real-time performance of LabVIEW for time-critical applications such as feedback control. The big advantage here is that you avoid having to learn any additional languages, such as those required for programming a PLC. You configure one or more LabVIEW RT systems that reside on a TCP/IP network and then exchange process information between those RT systems (known as **peer-to-peer** communications) or with the MMI machine. See Chapter 15, “LabVIEW RT,” for more information on LabVIEW RT.

**Single-loop controllers (SLCs)** do an admirable job of PID, on/off, or fuzzy logic control for industrial applications. They are generally easy to configure, very reliable, and many offer advanced features such as recipe generation and automatic tuning. Like PLCs, however, they lack the ability to record data, and it may not be feasible to locate the SLC close to the operator’s preferred working location. By using a communications link (usually RS-232), LabVIEW can integrate SLCs into a distributed process control system.

**Choose your system.** In summary, here are some important control system design parameters that might lead you to a first-cut decision as to whether LabVIEW (probably with DSC) is a practical choice for your application.

*Number of I/O points and number of network nodes.* What is the overall scale of your system? Up to a few hundred I/O points, even a stand-alone PC running LabVIEW has a good chance of doing the job. With somewhat higher point counts, high-speed requirements, or a need to have the I/O distributed over a wider area, PLCs offer lots of advantages, and LabVIEW makes a nice MMI. If you are tackling an entire plant, then LabVIEW might be suited to some local control applications, but a DCS or a large, integrated SCADA package may be a better choice. LabVIEW with DSC, for example, is probably suitable for installations with several thousand points.

*Planned expansion.* Continuing with the first topic, remember to include any plans for future expansion. A lab-scale pilot plant is ideal for LabVIEW, but its plant-scale implementation may be a whole ‘nother story. A poorly chosen control system may result from underestimation of the overall needs. Beware.

*Integration with other systems.* Needs for corporate-level computing will influence your choice of control computers and networks. For instance, if you live in a UNIX/X Window world, then a SPARC station or Linux box running LabVIEW might be a good platform.

*Requirements for handling unique, nonstandard instruments.* LabVIEW has tremendous advantages over commercial process

control software products when it comes to handling unusual instrument interfaces. Because you can easily write your own drivers in G, these special situations are a nonproblem. Similarly, DSC permits you to add ordinary LabVIEW drivers, running in parallel with the native Process Control server-based drivers.

*Ease of configuration and modification.* What plans do you have for modifying your process control system? Some system architectures and software products are rather inflexible, requiring many hours of reconfiguration for seemingly simple changes. Adding a new instrument interface to a large DCS may seemingly require an act of Congress, as many users have discovered. Meanwhile, LabVIEW remains a flexible solution, so long as you design your application with some forethought.

*Overall reliability.* It's hard to compare computers, operating systems, and software when it comes to reliability. Suffice it to say that simple is better, and that's why many process engineers prefer PLCs and SLCs: They are simple and robust. The MMI software is, in a sense, of secondary importance in these systems because the process can continue to operate (if desired) even while the MMI is offline. We've found that the system architecture is often more important than the brand of computer or software. Using **redundant** hardware is the ultimate solution where reliability is paramount. Many PLC and DCS vendors offer redundancy. Additionally, you should consider adding fail-safe mechanisms, such as watchdog timers, to insure safety and prevent major hazards and losses when things go awry. Software bugs, hardware failures, and system crashes can be very costly in process control.\*

*Existing company standards and personal biases.* Everyone from the CEO to the janitor will have opinions regarding your choice of hardware and software. For the small, experimental systems that we work on, LabVIEW has many advantages over other products. As such, it has become very popular—a definite positive bias—and with good reason. However, we would *not* use it to control a new hundred million-dollar facility, no matter how biased we might be.

---

\*A friend of ours at a major process control system manufacturer reports that Bill Gates and company have effectively lowered our expectations. Prior to the great rise in the popularity of Windows in process control, systems were based on VAX/VMS, UNIX, and very expensive custom software. If a system crashed in a plant operation, our friend's company would get a nasty phone call saying, "*This had better never happen again!*" Now, he finds that Windows and the complex application software come crashing down so often that plant operators hardly bat an eye; they just reboot. It's a sad fact, and we wonder where this is all going.

*Cost.* The bottom line is always cost. One cost factor is the need for system integration services and consultants, which is minimal with LabVIEW, but mandatory with a DCS. Another factor is the number of computers or smart controllers that you might need: A stand-alone PC is probably cheaper than a PC plus a PLC. Also, consider the long-term costs such as operator training, software upgrades and modifications, and maintenance. Plan to evaluate several process control packages—as well as LabVIEW with DSC—before committing your resources.

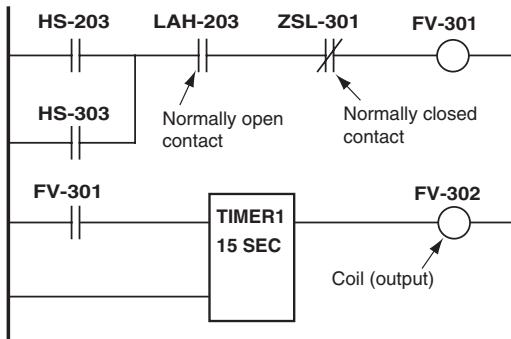
## Working with Smart Controllers

Adding smart controllers to your process control system is smart engineering. PLCs, SLCs, and other programmable devices are practical solutions to your real-world control problems. They are completely optimized to perform real-time continuous or sequential control with the utmost in reliability while managing to be flexible enough to adapt to many different situations. Take advantage of them whenever possible.

**Programmable logic controllers (PLCs).** A programmable logic controller (PLC) is an industrial computer with a CPU, nonvolatile program memory, and dedicated I/O interface hardware (Bryan and Bryan 1988). Conceived by General Motors Corporation in 1968 as a replacement for hardwired relay logic, PLCs have grown dramatically in their software and hardware capabilities and now offer incredible power and flexibility to the control engineer. Major features of a modern PLC are as follows:

- Modular construction
- Very high reliability (redundancy available for many models)
- Versatile and robust I/O interface hardware
- Wide range of control features included, such as logic, timing, and analog control algorithms
- Fast, deterministic execution of programs—a true real-time system
- Communications supported with host computers and other PLCs
- Many programming packages available
- Low cost

You can buy PLCs with a wide range of I/O capacity, program storage capacity, and CPU speed to suit your particular project. The simplest PLCs, also called *microcontrollers*, replace a modest number of relays and timers, do only discrete (on/off, or boolean) logic operations, and



**Figure 18.7** Ladder logic is a language for PLCs. Each item in the diagram corresponds to an emulated piece of hardware—contacts (switches), timers, relay coils, and so forth.

support up to about 32 I/O points. Midrange PLCs add analog I/O and communications features and support up to about 1024 I/O points. The largest PLCs support several thousand I/O points and use the latest microprocessors for very high performance (with a proportionally high cost, as you might expect).

You usually program a PLC with a PC running a special-purpose application. **Ladder logic** (Figure 18.7) is the most common language for programming in the United States, though a different concept, called **GRAFCET**, is more commonly used in Europe. Some programming applications also support BASIC, Pascal, or C, either intermixed with ladder logic or as a pure high-level language just as you would use with any computer system.

There are about 100 PLC manufacturers in the world today. Some of the major brands are currently supported by OLE for Process Control (OPC) and LabVIEW drivers. Check the National Instruments Web site for new releases.

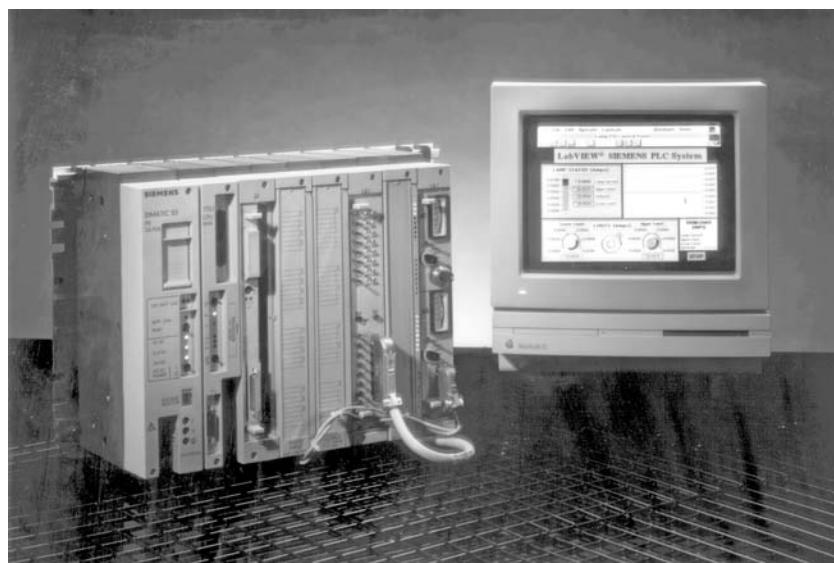
The ISA offers some PLC-related training you may be interested in. Course number T420, Fundamentals of Programmable Controllers, gives you a good overview of the concepts required to use PLCs. The textbook for the class, *Programmable Controllers: Theory and Implementation* (Bryan and Bryan 1988), is very good even if you can't attend the class. Another course, T425, Programmable Controller Applications, gets into practical aspects of system design, hardware selection, and programming in real situations.

**Advantages of PLCs.** Programmable controllers have become a favorite tool in the control industry because of their simplicity, low cost, rugged hardware, and reliable real-time performance. These are clear-cut advantages that will make them candidates for your project as well.

Interfacing couldn't be easier. Input and output modules are available for direct connection to 240-V ac and lower ac and dc voltages and a wide range of analog signals. Compare this with most interface systems which require extra isolation devices or adapters, especially for high voltages and currents.

Reliability may be the main reason for choosing a PLC over other solutions. Both the hardware and the operating software in a PLC are simple in nature, making it easier to prove correctness of design and simultaneously lowering the number of possible failure points. Contrast this with a general-purpose, desktop computer. How often does *your* computer hang up or crash? If your process demands predictable performance, day in and day out, better consider using a PLC. See Figure 18.8.

**PLC communications and register access.** Manufacturers of PLCs have devised many communications techniques and pseudostandard protocols. For instance, Modicon has its proprietary *Modbus* protocol, while Allen-Bradley has its *Data Highway Plus*—and the two are completely incompatible. The good news is, through an adapter module, any computer with RS-232 or Ethernet capability can communicate with these various data highways, assuming that you have the right driver software. All midrange and high-end PLCs have the ability to communicate



**Figure 18.8** PLCs like this Siemens model are popular in process control. SinecVIEW is a LabVIEW driver package that communicates with Siemens PLCs via a serial interface. (Photo courtesy of National Instruments and CITvzw Belgium.)

peer to peer, that is, between individual PLCs without host intervention. A variety of schemes exist for host computers as well, serving as masters or slaves on the network.

Once a PLC has been programmed, your LabVIEW program is free to read data from and write data to the PLC's registers. A register may represent a boolean (either zero or one), a set of booleans (perhaps 8 or 16 in one register), an ASCII character, or an integer or floating-point number. Register access is performed at a surprisingly low level on most PLCs. Instead of sending a message like you would with a GPIB instrument ("start sequence 1"), you poke a value into a register that your ladder logic program interprets as a command. For instance, the ladder logic might be written such that a 1 in register number 10035 is interpreted as a closed contact in an interlock string that triggers a sequential operation.

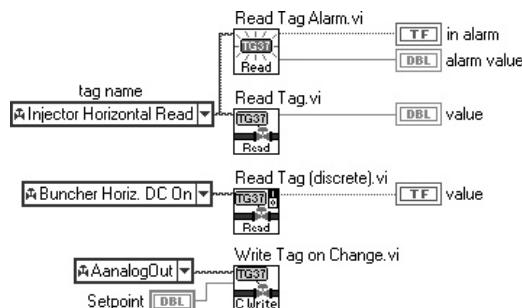
There are a few things to watch out for when you program your PLC and team it with LabVIEW or any other host computer. First, watch out for conflicts when writing to registers. If the ladder logic *and* your LabVIEW program both write to a register, you have an obvious conflict. Instead, all registers should be one-way; that is, either the PLC writes to them or your LabVIEW program does. Second, you may want to implement a **watchdog program** on the PLC that responds to failures of the host computer. This is especially important when a host failure might leave the process in a dangerous condition. A watchdog is a timer that the host must *hit*, or reset, periodically. If the timer runs to its limit, the PLC takes a preprogrammed action. For instance, the PLC may close important valves or reset the host computer in an attempt to reawaken it from a locked-up state.

PLC communications is another area where the Datalogging and Supervisory Control (DSC) module driver model has advantages over the much simpler LabVIEW driver model. With DSC, National Instruments supplies a large set of **OPC** drivers that connect to a wide variety of industrial I/O devices, including PLCs. Also, many hardware manufacturers supply OPC-compliant drivers that you can install and use with DSC. OPC is a layered, plug-and-play communications interface standard specified by Microsoft for use with all versions of Windows. Essentially, it provides a high-performance client-server relationship between field devices and user applications, including LabVIEW, Excel, Visual Basic, and many other packages. There's even an organization—the OPC Foundation—that promotes the OPC standards. Visit them at [www.opcfoundation.org](http://www.opcfoundation.org).

**An OPC-based PLC driver example.** When properly designed, OPC drivers are easy to configure and use, and add only modest overhead to the communications process. Included with DSC is the Industrial Automation

Servers CD with hundreds of OPC drivers that you install into your operating system and configure with a utility program that is external to LabVIEW. This basic configuration step tells the driver what communications link you're using (serial, Ethernet, etc.) and how many and what kind of I/O modules you have installed. Once this low-level step is performed, you work entirely within the features of the DSC tool set to configure and use individual **tags** (I/O points). The **Tag Configuration Editor** is the place where you create and manage tags. For a given tag, you assign a tag name, a PLC hardware address, scale factors, alarm limits, and trending specifications. For very large databases, you may find it easier to manage the information in a spreadsheet or a more sophisticated database program. In that case, the Tag Configuration Editor allows you to export and import the entire tag database as ASCII text. Once the tag database is loaded, the editor does a consistency check to verify that the chosen OPC server recognizes the hardware addresses that you've chosen.

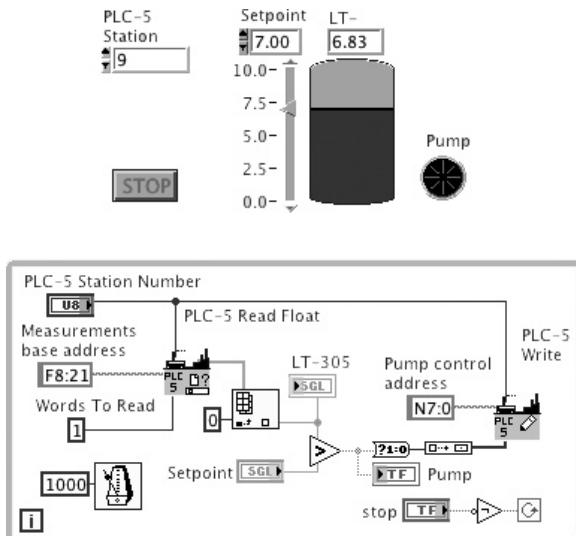
When you install DSC, LabVIEW gains a large number of control and function subpalettes. Among the functions are VIs to read, write, and access properties of tags—and it's really easy to do. Figure 18.9 shows a few of the basic VIs and how they're used for simple read/write activities. In most cases, you just supply a tag name and wire up the input or output value. The database and OPC drivers take care of the rest. You can also use **G Wizards**, which create blocks of G code on the diagram. A pop-up item available on every control and indicator brings up a dialog through which you assign a tag name and an action. For instance, if you want an indicator to display the value of an analog tag, a small While Loop with the Read Tag VI will appear on the diagram, already wired to the indicator. A large number of options are available such as alarm actions (color changes and blinking) as well as the frequency of update.



**Figure 18.9** Here are a few of the VIs included with DSC that allow you to read and write tag values that reside in the database

The OPC-based database has several features that improve performance. When you define a tag, you assign a **scan interval**. That's the interval at which the OPC driver goes out and interrogates the hardware for an input tag. The rest of the time, there is no I/O activity, thus reducing overhead for both your host computer and the PLC or other I/O hardware. VIs that read tag values, such as Read Tag, can be configured to sleep until a new value is available. For output tags, you can configure tags to send data to the hardware only when there's a change in value. Again, this drastically reduces I/O message activity and is a key to the excellent performance that we've seen in systems using DSC.

**LabVIEW PLC driver example: HighwayVIEW.** Of the PLC driver packages currently available, HighwayVIEW, from SEG, was the first. It supports Allen-Bradley models PLC-2, PLC-3, and PLC-5 and accepts all data types for both inputs and outputs. Communications are handled by a program that is installed in your computer's operating system (a Control Panel device on the Macintosh; a DLL under Windows). This communications handler is particularly robust, offering automatic retries if an error occurs. Figure 18.10 shows a simple example of HighwayVIEW in action as a simple on/off limit controller application. An



array of floating-point values is read by the PLC-5 Read Float VI. If the tank level is greater than the setpoint, a drain pump is turned on by setting a bit with the PLC-5 Write VI.

### Single-loop controllers (SLCs)

Single-loop controllers are an old favorite of process engineers. Years ago, they were pneumatically operated, using springs, levers, bellows, and orifices to implement an accurate PID algorithm. To this day, there are still many installations that are entirely pneumatic, mostly where an intrinsically safe control system is required. Modern SLCs are compact, microprocessor-based units that mount in a control panel. Some models can control more than one loop. They are compatible with common input signals including thermocouples, RTDs, currents, and voltages, with various ranges. The output may be an analog voltage or current or an on/off digital signal controlled by a relay or triac. Other features are alarms with contact closure outputs and ramp-and-soak recipe operation. (See Figure 18.11.)

The front panel of a typical SLC consists of an array of buttons and one or more digital displays. They are fairly easy to understand and use, though some operators are frustrated by the VCR programming syndrome where there are too many obscure functions and too many menu choices. This is where LabVIEW can enhance the usefulness of an SLC by simplifying the presentation of controls and indicators. Low-level setup information can be hidden from the user while being properly maintained by your program.

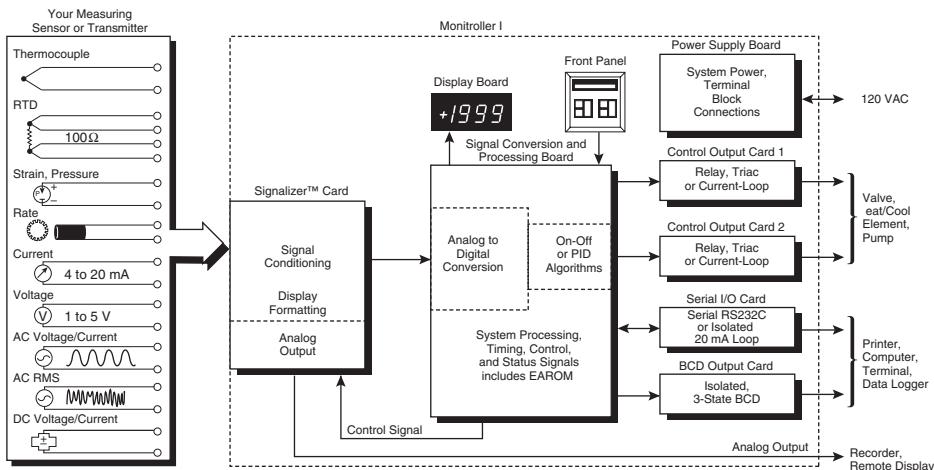
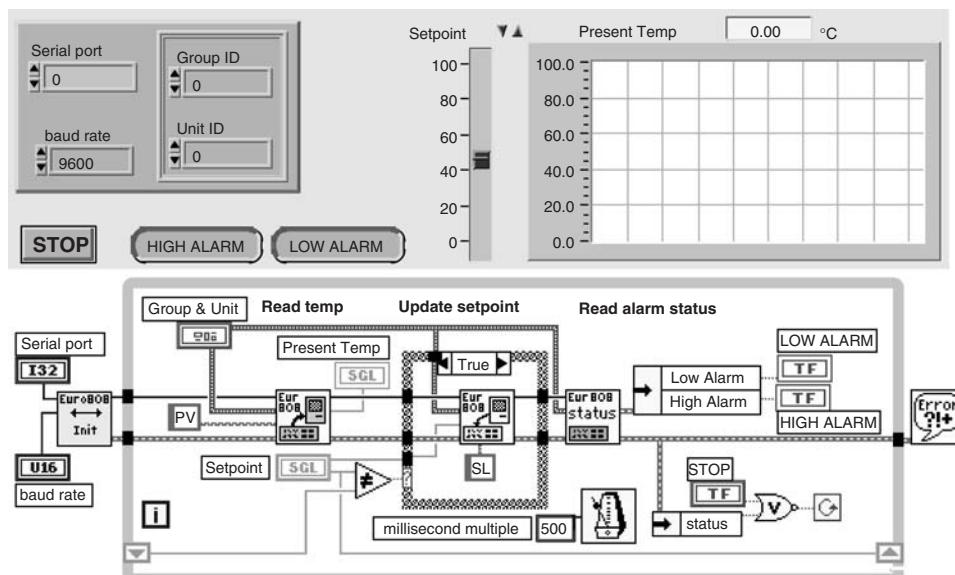


Figure 18.11 Block diagram of a typical single-loop controller. (Courtesy Analogic.)

All of the more sophisticated controllers offer communications options. Most use RS-232 or RS-422/485 serial, but at least one features GPIB (Research, Incorporated, with its Micristar controller). The communications protocols all vary widely, generally following no standard at all. One exception is a line of controllers from Anafaze that use the Allen-Bradley Data Highway protocol, the same one used by Allen-Bradley PLCs. Anafaze controllers are supported by AnaVIEW from SEG. As a bonus, some Anafaze models can handle up to 16 PID loops in one compact controller; they are more appropriately named **multiple-loop controllers**. A few other SLCs have LabVIEW drivers, such as the Eurotherm 808/847. (See Figure 18.12.) With a data highway, GPIB, or RS-422 party line, a multidrop network of controllers is feasible, permitting many stations to communicate with your LabVIEW system while consuming only one communications port.

The easiest way to control an SLC is with a good driver. A few controllers are available on the web. (e.g., Eurotherm), and a few are supported by commercial software vendors, as in the case of Anafaze. Consider the availability of drivers when choosing a controller; otherwise, you will have to write your own. A good driver package will permit you to read and write most any function in the controller.



**Figure 18.12** This example uses the Eurotherm 808/847 single-loop controller driver to update the setpoint and read the process variable and alarm status.

We really recommend SLCs when you have a modest number of control loops. Their high reliability and simplicity, combined with the power and flexibility of a LabVIEW MMI, makes a fine process control system.

### Other smart I/O subsystems

There are many other ways to distribute the intelligence in your process control system besides PLCs and SLCs. All you really need is a remote or distributed I/O subsystem with local intelligence and communications that can execute either sequential or continuous control algorithms.

Multifunction scanners are more often used as simple data acquisition units, but they contain a surprising amount of control functionality as well. You can download programs to do scanning and units conversion, limit checking, and some kinds of feedback control. You have many options with the many types of interface modules available for these scanners. Check for availability of LabVIEW drivers before deciding on a particular model.

An often overlooked option is to use LabVIEW-based slave systems with little or no MMI functionality. The slave nodes communicate with the main MMI node, exchanging data in some LabVIEW formats that you decide upon. Slaves handle all the real-time data acquisition and control activity without the performance penalties of a busy graphical display or lots of user intervention. This technique is advantageous when you need sophisticated algorithms that a PLC or other simple processor can't handle. Also, since everything is written in LabVIEW, it's easier to maintain. Networking techniques are discussed later in this chapter.

One downside to using LabVIEW in the loop *may* be in the area of reliability, depending upon how critical your tasks are. Where I work, it is a policy never to use a general-purpose computer for safety interlocks.

PLCs, on the other hand, are considered acceptable. The differences are in the ruggedness of the hardware and in the simplicity of the PLC software, which simply doesn't crash and which also has a very deterministic response time. The same cannot be said of a typical LabVIEW system running on top of a complex operating system. You will have to be the judge on this matter of safety and reliability.

## Man-Machine Interfaces

When it comes to graphical man-machine interfaces, LabVIEW is among the very best products you can choose. The library of standard and customizable controls is extensive; but more than that,

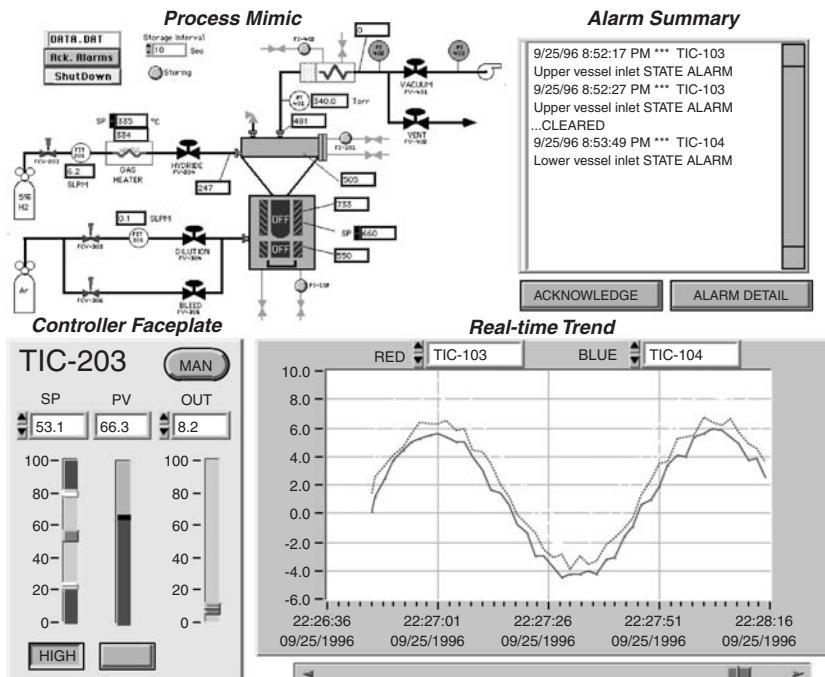


Figure 18.13 Some of the basic process displays that you can make in LabVIEW.

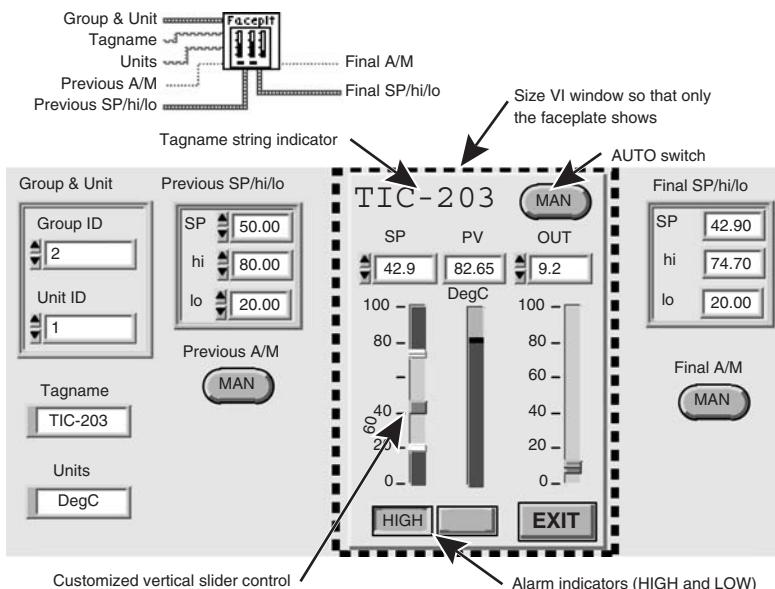
the function of each control is determined by you, the crafty programmer. In the process control world, there are several commonly used displays that you will probably want to implement in one form or another (Figure 18.13):

- **Process mimic displays** are based on simplified P&ID diagrams. Various elements of the display are animated, tying measurements and status information directly to physical elements in the plant. Operators really like this kind of display. You can draw a representation of the system and import the picture into LabVIEW, then overlay it with various controls and indicators. Valves, heaters, and alarm indicators are usually booleans, while various numeric controls and indicators are placed near their respective instruments. DSC provides a library of process control symbols that speed the creation of your displays.
- **Trending displays** (graphs or strip charts), of which there are two types: historical trends and real-time trends. A real-time trend displays up-to-the-minute data, but may not go very far back in time.

It relies primarily on data stored in memory. Historical trends usually read and display data from large disk files. Trending is discussed later in this chapter.

- **Controller faceplate displays** look and act much like the front panels of SLCs. They permit the operator to observe and control all the settings of PID and other types of analog controllers. Digital I/O or alarm status can also be presented in the form of boolean indicators, like the example in Figure 18.14.
- **Alarm summary displays** give the operator a concise overview of important status information, particularly I/O points that are out of specification in some way. You can use scrolling string indicators or tables to list recent messages or boolean indicators as a kind of status panel.

As an example, we decided to create a nice controller faceplate display that uses some of LabVIEW's more advanced features. This one mimics some of the commercial SLC front panels and is typical of the faceplates we've seen displayed on commercial DCS screens. (On a DCS, they usually put 8 or 10 faceplates on one screen, and the faceplate is a standard indicator element.) Figure 18.14 shows the panel of the controller subVI. You would normally size the VI window such that



**Figure 18.14** Here's the panel of a subVI that mimics a single-loop controller faceplate. You can call it from other VIs to operate many controllers since everything is programmable.

only the faceplate part of the panel is visible; the other items are just parameters for the calling VI. This subVI is programmable in the sense that the tag name, units, and other parameters are passed from the caller. This fact permits you to use one subVI to operate many control loops. The VI must be set to *Show front panel when called*.

The SP (setpoint) slider control is customized via the various pop-up options and the **Control Editor**. We first added two extra sliders by popping up on the control and selecting *Add Slider*. They serve as high and low alarm limits. We set the Fill Options for each of these alarm sliders to *Fill to Maximum* (upper one) and *Fill to Minimum* (lower one). In the Control Editor, we effectively erased the digital indicators for the two alarm limits by hiding them behind the control's main digital display.

The diagram in Figure 18.15 is fairly complex, so we'll go through it step by step. It illustrates the use of Local variables for control initialization. The I/O hardware in this example is, again, the Eurotherm 808 SLC. This VI relies on the controller to perform the PID algorithm, though you could write a set of subVIs that perform the same task

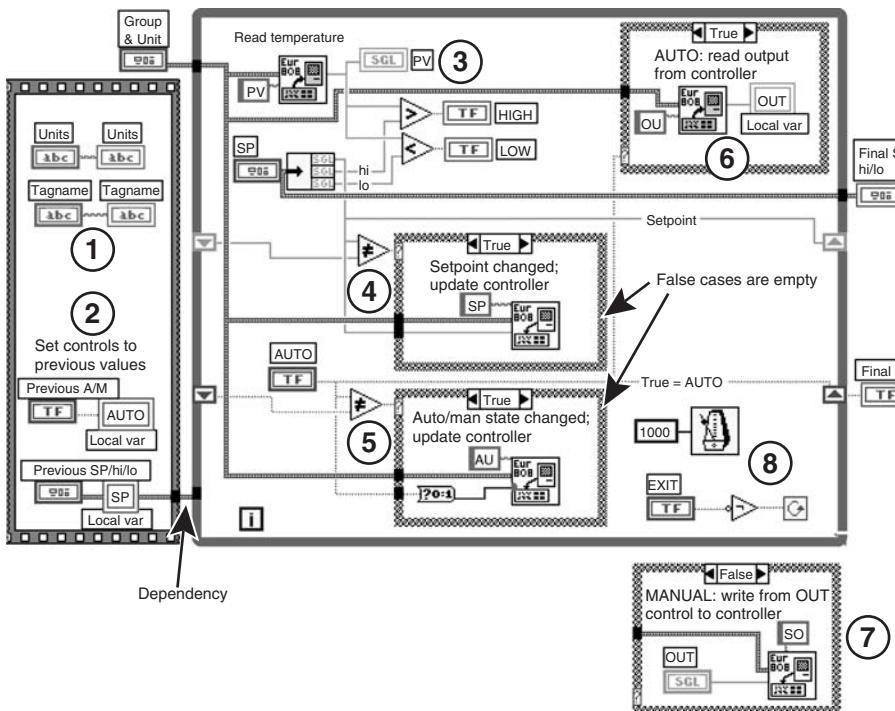


Figure 18.15 Diagram for the faceplate controller.

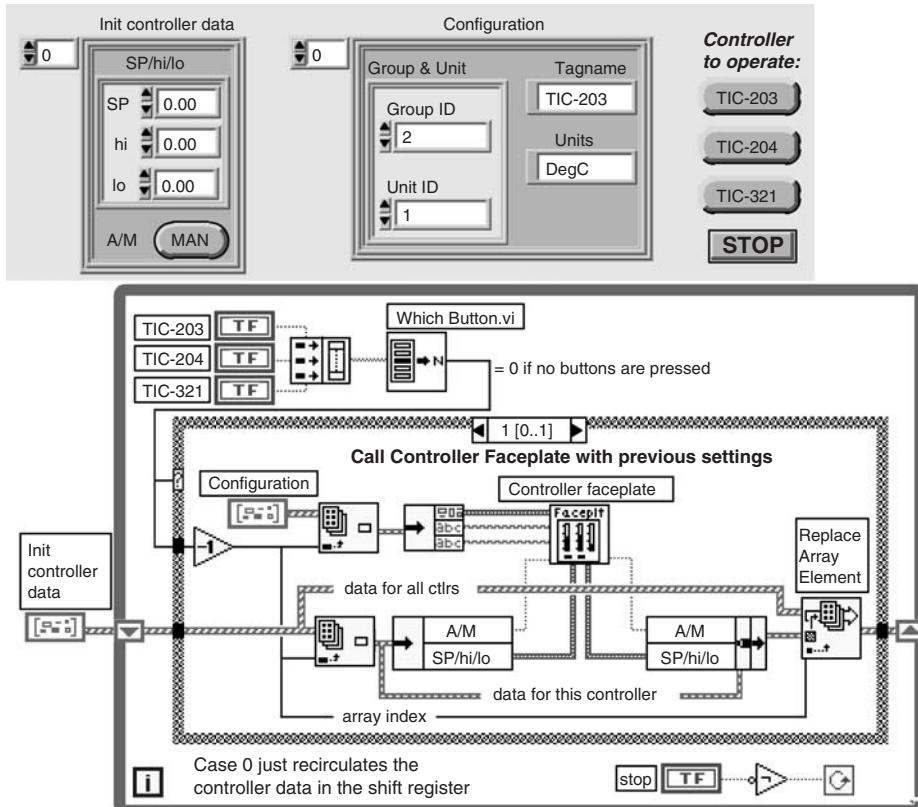
with the PID Control Toolkit in LabVIEW, in conjunction with any type of I/O.

1. **Tag name** and **Units** are copied from incoming parameters to indicators on the faceplate. This gives the faceplate a custom look—as if it was written just for the particular channel in use.
2. Previous settings for **Auto?** (the auto/manual switch) and **SP** (the setpoint slider) are written to the user controls on the faceplate by using local variables. This step, like step 1, must be completed before the While Loop begins, so they are contained in a Sequence structure with a wire from one of the items to the border of the While Loop.
3. The process variable (current temperature) is read from the controller, checked against the alarm limits, and displayed.
4. The current value for the setpoint is compared against the previous value in a Shift Register. If the value has changed, the setpoint is sent to the controller. This saves time by avoiding retransmission of the same value over and over.
5. In a similar fashion, the **Auto?** switch is checked for change of state. If it has changed, the new setting is sent to the controller.
6. If the mode is automatic, the **Out** (output) control is updated by using a Local variable. The value is read from the controller.
7. If the mode is manual, **Out** supplies a value that is sent to the controller. These two steps illustrate an acceptable use of read-write controls that avoids race conditions or other aberrant behavior.
8. The loop runs every second until the user clicks **Exit**, after which the final values for **Auto?** and **SP/hi/lo** are returned to the calling VI for use next time this VI is called.

Figure 18.16 shows how this subVI might be used in an application to support multiple controllers. The panel contains a configuration array that identifies each controller, a set of buttons to call the *desired controller*, and an initialization array. When one of the buttons is pressed, our old friend, the **Which Button** VI, returns a nonzero value and Case 1 is executed (Case 0 just recirculates data in the Shift Register, waiting for a button to be pressed).

Inside the Case, the **Configuration** cluster array is indexed to extract the setup information for the selected controller, which is then passed to the Controller Faceplate subVI, which opens when called.

Similarly, the controller data array, circulating in the Shift Register, is indexed and unbundled. When the Controller Faceplate subVI finishes, the data it returns is bundled and the current element in the



**Figure 18.16** An application that uses the Faceplate Controller subVI. The Shift Register acts as a database for previous controller values.

data array is replaced. The next time this controller is called, data from the previous call will be available from the Shift Register. Note the use of **Bundle by Name** and **Unbundle by Name**. These functions show the signal names so you can keep them straight.

This overall procedure of indexing, unbundling, bundling, and replacing an array element is a versatile database management concept that you can use in configuration management. Sometimes the data structures are very complex (arrays of clusters of arrays, etc.), but the procedure is the same, and the diagram is very symmetrical when properly laid out. Little memory management is required, so the operations are reasonably fast. One final note regarding the use of global memory of this type. All database updates must be controlled by one VI to serialize the operations. If you access a global variable from many locations, sooner or later you will encounter a race condition where two callers attempt to write data at the same time. There is no way of knowing

who will get there first. Each caller got an original copy of the data at the same time, but the last one to write the data wins. This is true for both the built-in LabVIEW globals and the one based on Shift Registers that you build yourself.

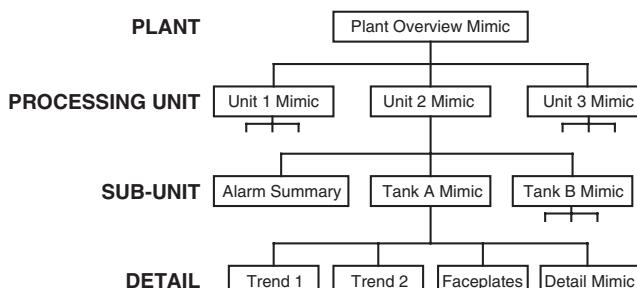
If you want to save yourself lots of effort, check out the DSC **MMI G Wizard**. It lets you interactively create your MMI from a library of standard display elements such as trends, faceplates, and alarms.

What's amazing is that you don't have to do any G programming at all, if you don't want to.

### Display hierarchy

Naturally, you can mix elements from each of the various types of displays freely because LabVIEW has no particular restrictions. Many commercial software packages have preformatted displays that make setups easy, but somewhat less versatile. You could put several of these fundamental types of displays on one panel; or better, you may want to segregate them into individual panels that include only one type of display per panel. A possible hierarchy of displays is shown in Figure 18.17. Such a hierarchy relies heavily on global variables for access to information about every I/O point and on pop-up windows (VIs that open when called).

Your displays need to be organized in a manner that is easily understood by the operator and one that is easy to navigate. A problem that is common to large, integrated control systems is that it takes too long to find the desired information, especially when an emergency arises. Therefore, you must work with the operators so that the display hierarchy and navigation process make sense. The organization in Figure 18.17 is closely aligned with that of the plant that it controls. Each subdisplay gives a greater level of detail about, or a different view of, a particular subsystem. This method is generally accepted by operators and is a good starting point.

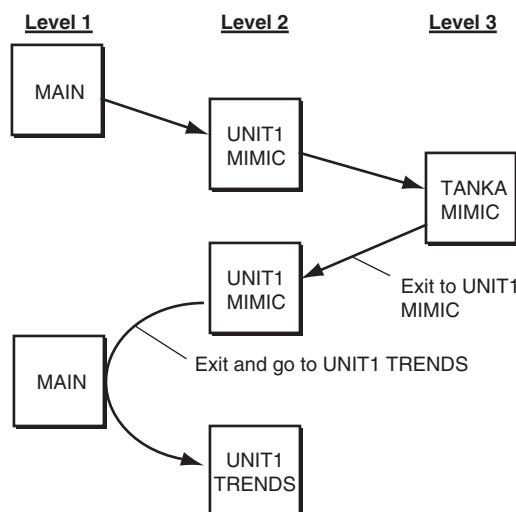


**Figure 18.17** Hierarchical process displays allow the operator to see any level of detail without having overcrowded screens.

The VI hierarchy can be mapped one for one with the display hierarchy by using pop-up windows (subVIs set to **Show front panel when called**). Each VI contains a **dispatcher** loop similar to the one shown later in this chapter (in Figure 18.29). Buttons on the panel select which display to bring up. A Case structure contains a separate display subVI in each frame. The utility VI, **Which Button**, multiplexes all the buttons into a number that selects the appropriate case. The dispatcher loop runs in parallel with other loops in the calling VI.

Each subVI in turn is structured the same way as the top-level VI, with a dispatcher loop calling other subVIs as desired. When it's time to exit a subVI and return to a higher level, an exit button is pressed, terminating the subVI execution, closing its panel, and returning control to the caller. An extension of this exit action is to have more than one button with which to exit the subVI. A value is returned by the subVI indicating where the user wants to go next (Figure 18.18). The caller's dispatcher is then responsible for figuring out which subVI to call next.

Making the actual buttons is the fun part. The simplest ones are the built-in labeled buttons or, you can paste in a picture that is a little more descriptive. Another method is to use a transparent boolean control on top of a graphic item on a mimic display, such as a tank or reactor. Then all the operator has to do is click on the tank, and a predefined display pops up. This is a good way to jump quickly



**Figure 18.18** Traversing the display hierarchy. When you click an exit button on a subdisplay VI, it returns a value that tells the calling VI where to go next.

to an associated trend, alarm, or other detail display. Remember that the operator has to *know* that the buttons are there because he or she can't *see* them. Therefore, you must be consistent in your use of such buttons, or the operators will get confused. To make a transparent boolean, select a simple boolean control such as the Flat Square Button, and use the Coloring tool to make both the foreground and background transparent (T).

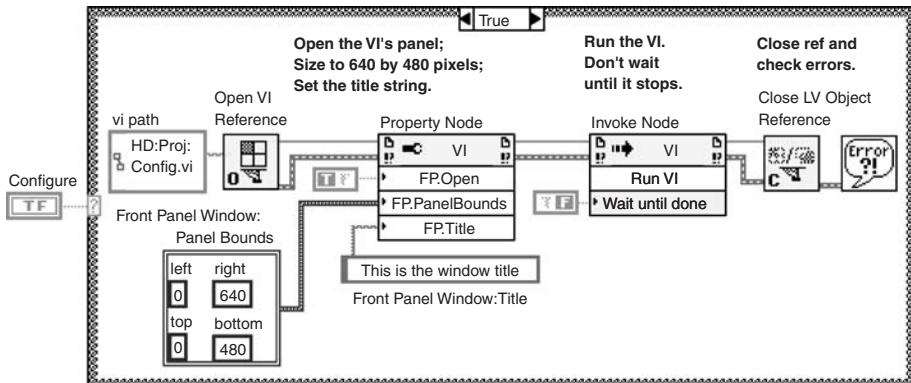
**Power Windows: Use the VI Server.** You can do some really interesting display tricks (and a lot more) with the **VI Server** functions, which you find in the **Application Control** function menu. They provide ways to manipulate VIs that go way beyond the regular VI Setup items, such as *Show front panel when called*.

The VI Server lets you act on VIs in three basic ways:

- **Property Nodes** change or interrogate attributes of a VI, such as the appearance of the front panel and VI information. *Note:* This is not the same kind of Property Node (formerly known as Attribute Nodes) that you use in conjunction with controls and indicators.
- **Invoke Nodes** cause a VI to take some kind of action, such as running, saving, or printing. This is also called *invoking a method* (one of those object-oriented programming terms).
- **Call By Reference Nodes** allow you to run a VI—and pass parameters to and from it—without the actual VI being wired into the diagram. This enables you to dynamically choose and run VIs while the application is running.

The VI Server uses a call-by-name approach, where you supply a string or path containing the exact name of the VI you wish to access to the **Open VI Reference** node. Once the VI is loaded into memory, you can start manipulating properties and methods.

Let's start with the situation where you need to change the mode of operation of your application in a significant way and you would prefer not to have all the required VIs in memory at all times. For instance, when the user clicks the Configure button, you can load the configuration manager from disk, rather than having it memory-resident at all times. Figure 18.19 shows a simple solution. We begin by opening a VI reference, which locates the Config VI and returns a VI refnum. Next, a Property Node opens the VI's front panel (but doesn't run it), sets the window size and position, and changes the title. An Invoke Node then runs the VI, and finally the reference is closed. This leaves the Config VI running. When it terminates, you can force its window to close and throw the Config VI out of memory by adding a bit of VI



**Figure 18.19** The VI Server opens a VI, sets the appearance of the panel, runs the VI, and leaves it running. It's a kind of remote control, and it works even over a network.

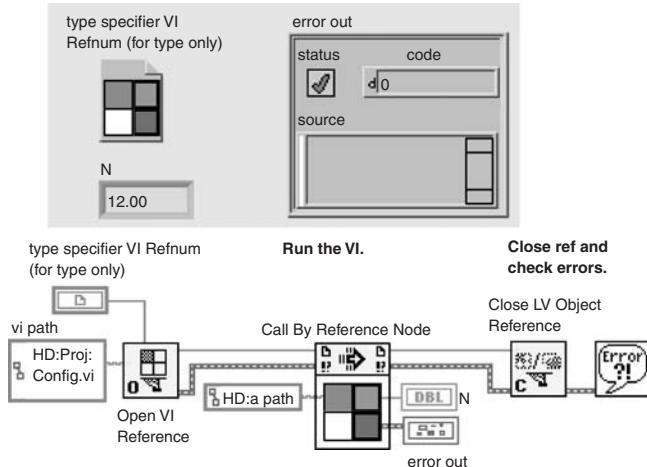
Server code to its diagram. All you need to do is have it open a VI reference (to itself), then have a Property Node set the Front Panel Open property to False.

The Call By Reference node operates in a manner similar to wiring the designated subVI directly into the diagram, right down to passing parameters to controls and receiving results back from indicators on the subVI. The important difference is that the subVI is dynamically loaded and unloaded, and *which* subVI is loaded is determined by the path name that you supply. In order to make it possible for you to dynamically substitute “any” VI into the Call By Reference node, you must make the connector pane identical in all of the candidate VIs.

Next, you have to tell the VI Server what your standard connector pane looks like. Beginning with an Open VI Reference function, pop up on the **type specifier VI** input and choose **Create Control**. A VI Refnum control will appear on the panel. Drag the icon of one of your standard VIs onto that VI Refnum control, and the connector pane will appear there. You have now created a *class* of VIs for the VI Server.

Drop a Call By Reference Node on the diagram, connect it to the Open VI Reference function, and you’ll see your connector pane appear. Then all you have to do is wire to the terminals as if the standard VI were actually placed on the diagram. Figure 18.20 shows the complete picture. In this example, the selected subVI runs in the background without its panel open unless it’s configured to Show Front Panel When Called. You can also add Property Nodes to customize the appearance, as before.

The VI Server offers additional opportunities for navigation among display VIs. To speed the apparent response time of your user interface when changing from one MMI window to another, consider leaving several of the VIs running at all times, but have only the currently active



**Figure 18.20** The Call By Reference node loads a VI from disk and exchanges parameters with it as if the subVI were wired directly into the diagram. This can save memory since the subVI is loaded only when required.

one visible. When the user clicks a button to activate another window, use an Invoke Node to open the desired panel. Its window will pop to the front as fast as your computer can redraw the screen. A similar trick can prevent users from getting lost in a sea of windows. If your display hierarchy is complex and you allow more than one active window at a time, it's possible for the user to accidentally hide one behind the other. It's very confusing and requires excessive training to keep the user out of trouble. Instead, try to simplify your display navigation scheme, or use Invoke Nodes to assist the user.

### Other interesting display techniques

You can customize controls and indicators by replacing the built-in pictures with custom ones from a drawing package. A useful example, submitted by Corrie Karlsen of LLNL, is shown in Figure 18.21. The system being controlled has a carousel with 10 sample bottles and a liquid sampling valve. When the operator moves the valve icon, a stepper motor moves the carousel to the appropriate bottle. The operator presses a **fill** button (not shown), and the valve is cycled open for a predetermined period of time and then closed. This LabVIEW indicator then responds by updating the full/empty status of the sample bottles, which are boolean indicators.

You can create boolean controls that look like valves with different open and closed states by pasting in a pair of pictures. If the operator prefers feedback regarding the status of a valve that has limit switches,



**Figure 18.21** The sample valve is a horizontal slider control with the valve picture pasted in as the handle. The control has no axis labels, and the housing is colored transparent. The bottles are boolean indicators, and the pipes are static pictures.

you can place a **Pict Ring** indicator for status on top of a transparent boolean control for manual actuation. For a valve with high and low limit switches, the ring indicator would show open, transit, closed, and an illegal value where both limit switches are activated.

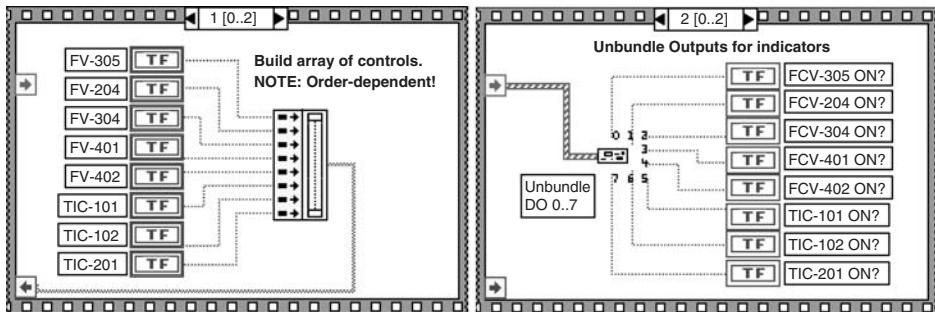
You can animate pipes, pumps, heaters, and a host of other process equipment by using picture booleans or Pict Rings. To simulate motion, create a picture such as a paddle wheel in a drawing program, then duplicate it and modify each duplicate in some way, such as rotating the wheel. Then, paste the pictures into the Pict Ring in sequence. Connect the ring indicator to a number that cycles through the appropriate range of values on a periodic basis. This is the LabVIEW equivalent of those novelty movies that are drawn in the margins of books.

Perhaps the most impressive and useful display is a process mimic based on a pictorial representation of your process. A simple line drawing such as a P&ID showing important valves and instruments is easy to create and remarkably effective. If your plant is better represented as a photograph or other artwork, by all means use that. You can import CAD drawings, scanned images, or images from a video frame grabber to enhance a process display.

**Property nodes** allow you to dynamically change the size, position, color, and visibility of controls and indicators. This adds a kind of animation capability to LabVIEW. The Blinking attribute is especially useful for alarm indicators. You can choose the on/off state colors and the blink rate through the LabVIEW Preferences.

### Handling all those front panel items

Process control panels tend to be loaded with many controls and indicators. This can make your diagram very busy and hard to understand. What you need to do is bundle related controls into arrays or



**Figure 18.22** Clean up your diagram by combining controls and indicators into arrays or clusters. Other frames in this sequence are the sources and sinks for data. The unbundler subVI on the right extracts boolean values from an array or cluster. You could also use Unbundle By Name.

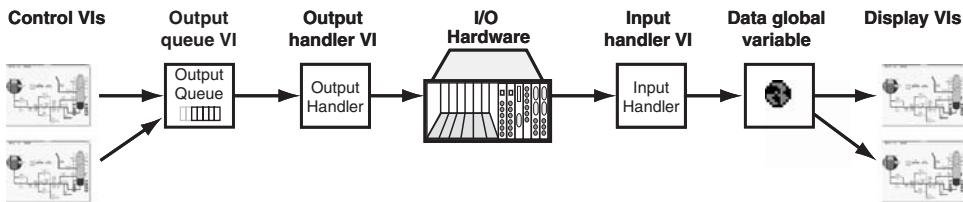
clusters and pass them to subVIs for processing. Similarly, indicators are unbundled for display after being returned from source VIs (Figure 18.22). A Sequence structure helps to conserve screen space. It doesn't impair the visual flow of the program too much because only a single item is passed from one frame to the next.

One hazard in this array-based scheme is that all the controls and indicators are order-dependent; that is, element three of the control array is always PV-401. Any wiring or indexing errors anywhere in your program will send the wrong value. An alternative is to use clusters and the Bundle by Name and Unbundle by Name functions. There is still a limitation with named clusters, however; you can't arbitrarily configure channels (insert or delete them) during execution with a configuration manager. Channels that you access must always exist, and editing of one or more diagrams will always be necessary to change the overall configuration. The way out of this quandary is a *real-time database*, a topic we cover later in this chapter.

Experts in human factors tell us that having too much information on one screen leads to information overload. Operators can't find what they're looking for, and the important values get lost in a sea of colored lights. If you find that your MMI display VI has a ridiculous number of controls and/or indicators, it's time to break it into multiple panels. It will then be easier for operators to use, and easier for the programmer to design and maintain. This is just one more reason for using hierarchical displays.

## Data Distribution

If you think about MMI display hierarchies, one thing you may wonder about is how the many subVIs exchange current data. This is a data distribution problem, and it can be a big one. A complex process



**Figure 18.23** Data distribution in a simple LabVIEW process control system. Control VIs write new settings to an output queue, from which values are written to the hardware by an output handler VI. An input handler VI reads data and stores it in a global variable for use by multiple display VIs.

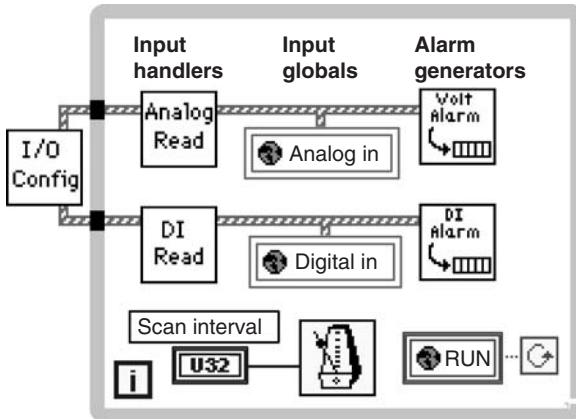
control system may have many I/O subsystems of different types, some of which are accessed over a network. If you don't use a coherent plan of attack, performance (and perhaps reliability) is sure to suffer. In homemade LabVIEW applications, global variables generally solve the problem, if used with some caution.

Most commercial process control systems, including DSC, use a **real-time database** to make data globally accessible in real time. You can emulate this in your LabVIEW program at any level of sophistication. The general concept is to use centralized, asynchronous **I/O handler tasks** to perform the actual I/O operations (translation: stand-alone VIs that talk to the I/O hardware through the use of LabVIEW drivers). Data is exchanged with the I/O handlers via one or more global variables or queues (Figure 18.23). User-interface VIs and control VIs all operate in parallel with the I/O handlers.

### Input scanners as servers

The **client-server** model is very effective for the input data in process control systems. An input handler task, a server VI which I like to call a **scanner**, periodically polls all the defined input channels, applies scale factors, and writes the results to global variables. The scanner is also a logical place to check any alarm limits. Depending on your application's complexity, the scanner VI may also be responsible for some aspects of trending or data archiving.

Figure 18.24 is a simplified scanner VI. It is called by a top-level VI and runs periodically (in parallel with the top-level VI) until a boolean global variable called *RUN* is set to False. Two input handler VIs—one for analog inputs and the other for digital—acquire raw data, filter it, and scale it to engineering units. Handler outputs are cluster arrays containing each channel's value, its name, and any other information that client tasks might need. This information is written to global variables and passed to alarm generator VIs that test each channel for alarm limit violations. Alarm flags or messages are stored in a queue (see the section entitled "Using an Alarm Handler," which follows).



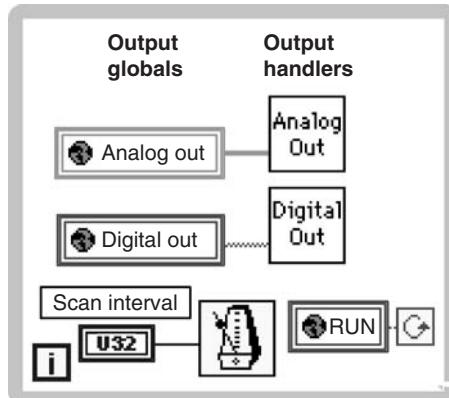
**Figure 18.24** A scanner VI that handles analog and digital inputs. The I/O configuration subVI, at left, passes information to the input handlers. Data from the input handlers is written to other global variables for use by client VIs. Alarm limits are also checked, and the results are written to alarm message queues.

As you might expect, process control has many of the same configuration needs as a data acquisition system. Each input handler requires information about its associated input hardware, channel assignments, and so forth. This information comes from an I/O configuration VI, which supplies the configurations in cluster arrays. If configuration information is needed elsewhere in the hierarchy, it can be passed directly in global variables, or it can be included in the data cluster arrays produced by the input handlers.

Here's an important tip regarding performance. Where possible, avoid frequent access of string data, particularly in global variables. Strings require extra memory management, and the associated overhead yields relatively poor performance when compared to all other data types. Not that you should *never* use strings, but try to use them sparingly and only for infrequent access. For instance, when you open an operator display panel, read channel names and units from the configuration database and display them just once, not every cycle of a loop.

### Handling output data

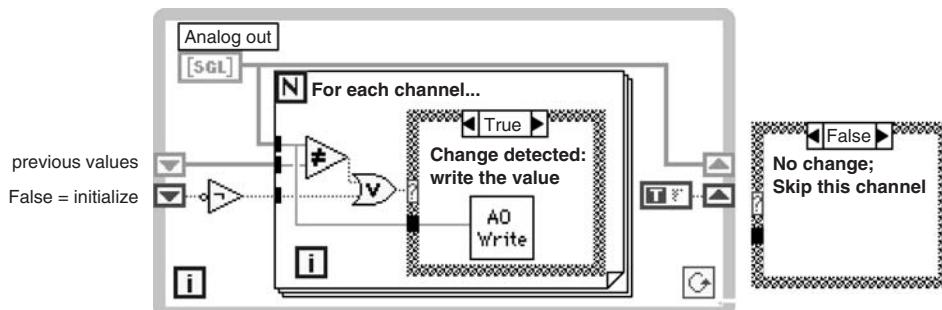
For efficiency, it may be better to update an output only when its value changes because some devices have a fair amount of overhead associated with communications. Like the input model we've been discussing, you can use an output handler and an output global variable that contains the values for each output channel. The global variable carries an array with each element of the array corresponding to an



**Figure 18.25** An output scanner, which is quite similar to an input scanner.

output channel. Any control VI can update values in the output global. The output handler checks for changes in any of the output channels and updates the specific channels that have changed. You can combine the handler and its associated global variable into an output scanner, as in Figure 18.25, which works like an input scanner in reverse.

To test for changes of state on an array of output values, use a program similar to the one in Figure 18.26. A Shift Register contains the values of all channels from the last time the handler was called. Each channel's new value is compared with its previous value, and if a change has occurred, the output is written. Another Shift Register is used to force all outputs to update when the VI is loaded. You could also add a **force update** or **initialize** boolean control to do the same thing. Depending upon the complexity of your system, channel configuration



**Figure 18.26** An output handler that only writes to an output channel when a change of value is detected. The upper Shift Register stores the previous values for comparison, while the lower Shift Register is used for initialization. Its value is False when the VI is loaded, forcing all outputs to update. Additional initialization features may be required if the program is stopped then restarted.

information may be required to perform the output write operation. If so, you can read the configuration global variable and pass that information along to the subVI that writes the data.

Because the Analog Output global variable contains an *array* of values, control VIs have to *update* that array, not overwrite it. Updates are easy to do. Just read the array from the global, use **Replace Array Element** to surgically change the value of a single channel, then write the array back to the global. Please note that there is again a great probability of a race condition arising. If the global variable is accessed by multiple callers, they will clash over who writes last. The solution is to encapsulate all *write* operations for a global variable inside a subVI, effectively serializing access. There is, of course, no limit to the number of VIs that can simultaneously read the global data.

Another way to avoid race conditions is to use an output **queue**. A queue is a first-in, first-out, or FIFO, construct that works like people standing in line. When a high-level VI needs to update an output, it calls an intermediate VI that adds the request to an output queue. Later, the output handler VI reads and empties the queue and updates each specified output.

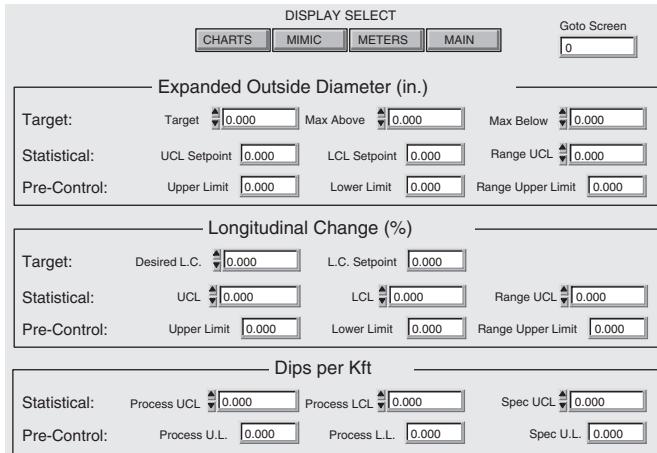
The Datalogging and Supervisory Control (DSC) module includes a very capable process control database,\*—one of the main reasons that you should consider using this package for your more complex applications. It's actually written in G (though they don't supply the diagrams for you to fiddle with) and the tag access VIs make it very easy to access information by tag name. All of the difficult synchronization problems, especially with outputs, are taken care of for you. The database interfaces with drivers based on the OPC server model, giving you a consistent and transparent connection with all of your I/O points. Inputs are scanned at rates that you choose, alarms are checked, and outputs are updated when a change is detected. It works even with other networked PCs running DSC, exchanging input and output data with remote I/O as if it was on the local machine.

### Display VIs as clients

By now you should be getting the idea that global variables are key elements when you are designing a versatile process control system. VIs that display data are very easy to write because global variables

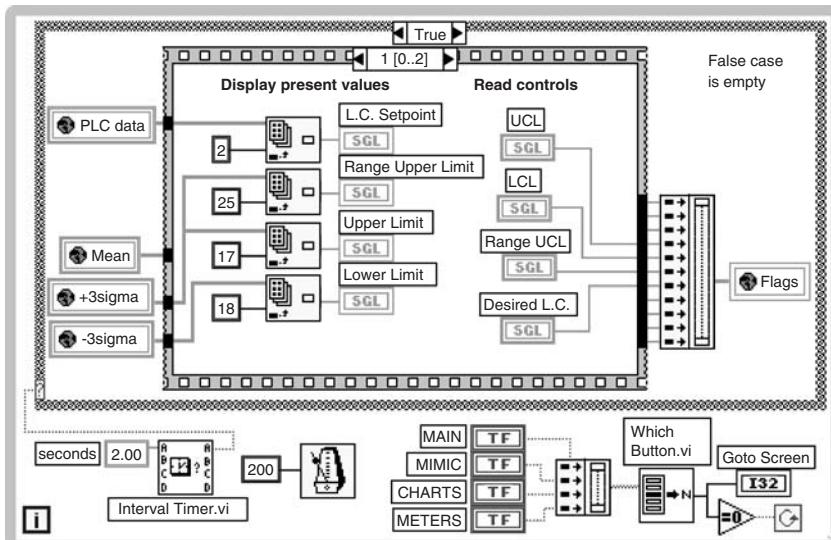
---

\* *Historical note:* In early 1991, Gary wrote a letter to the LabVIEW developers describing the elements and behaviors of a process control database, hoping that they would build such a beast into LabVIEW. As National Instruments got more involved with the process control industry, they realized how important such a capability really was. Audrey Harvey actually implemented the first prototype LabVIEW SCADA package in LabVIEW 3 sometime in 1993. It eventually became the basis for DSC, now available as DSC.



**Figure 18.27** Front panel of a display and control VI that is used to monitor and set operating specifications.

solve the data distribution problem. The input scanner supplies up-to-date values for all inputs, while the configuration part of your system supplies such things as channel names, units, and other general information, all available through global variables. Figures 18.27 and 18.28 show how we wrote a typical display/control VI by using the methods



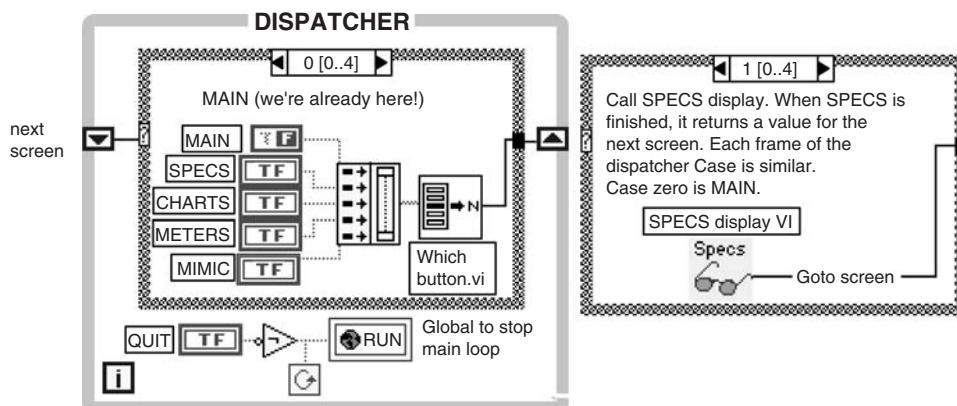
**Figure 18.28** Diagram for the display/control VI. The Case structure and interval timer limit how often the displays and controls are updated. A Sequence logically groups the controls and indicators, eliminating clutter. The VI loops until the user clicks a button to jump to another display.

we've discussed. Since this display example (Figure 18.27) is part of a hierarchical suite, there are buttons marked *Display Select* near the top. Clicking one of them causes this VI to terminate and returns an appropriate value in **Goto Screen** (which would normally be positioned off-screen so the user doesn't see it). The rest of the panel is a collection of numeric controls and indicators, grouped by function, with the help of some decorations. A fancy process mimic graphic could also be placed underneath the controls.

In the associated diagram in Figure 18.28, you can see that the **Which Button** VI determines whether the VI will continue to run, based on the state of all the Display Select buttons. The While Loop runs at a fairly fast rate, on the order of every 200 ms or so, to provide rapid response to the buttons. It would be wasteful, however, to update all the display items at that rate, since the data is only updated by the scanners every couple of seconds. The **Interval Timer VI** controls a Case structure that contains all the real activity. Every two seconds, the Case is True; otherwise, the False case, which contains nothing, executes.

Input data comes from several global variables containing data from a PLC and some computed statistics which are written by the input scanner VI. Output data from controls on the panel are built into an array and written to another global variable called *Flags*, which is used elsewhere to indicate system status. This is the only VI in the hierarchy that writes to *Flags* global variable, so there is no reason to create a queue or other protective device for that global variable. For clarity, we used a Sequence structure to logically group the front panel items.

If the display needs to call lower-level displays, add a second, independent While Loop as a **dispatcher**, as shown in Figure 18.29, just like the top-level VIs have. Some buttons will then activate the



**Figure 18.29** The dispatcher loop in a higher-level VI that calls the display VI from the preceding figures.

lower-level displays. Other buttons that cause a return to higher-level displays are wired as shown in Figure 18.28. This dispatcher concept is flexible and easy to program.

**Initializing controls.** All controls present on your display panel have to be initialized when the VI is called. Otherwise, they will revert to their default values, which will then be accepted as if those were user inputs—probably not a great idea. The simplest way to initialize controls is to supply initial values from the calling VI through the connector pane. Every control will have to be wired to the connector pane, and that may be a problem: You have only 20 terminals, one of which is probably the Goto Display output. If you have more than 19 controls to initialize, some will have to be colocated in clusters. Think about your panel layout and figure out which controls might be logically grouped in this way. You can effectively hide the border of a cluster by coloring it transparent so that it disappears behind the other graphical elements of your display.

You can also initialize controls with local variables from within the display VI. For the present example, initial values for the manual flag controls are easy to obtain because each control's previous value was written to the Flags global variable. After indexing each value out of the Flags array, use local variables to set each control's value, then start the main While Loop.

### Using network connections

To create a distributed control system (even a very small one), you will need to use a network. The most versatile network connections use Ethernet and TCP/IP, which is directly supported by LabVIEW through VIs in the Networking function palette. **IP (Internet Protocol)** performs the low-level packaging of data into *datagrams* and determines a network path by which messages can reach their destination. However, it offers no handshaking and doesn't even guarantee delivery. **TCP (Transmission Control Protocol)** adds handshaking and guarantees that messages arrive in the right order, making it quite reliable. **UDP (Universal Datagram Protocol)** is similar to TCP, but does no handshaking, making it somewhat faster and also offering the possibility of broadcasting a message to many recipients. You establish sessions between each LabVIEW system on the network, then exchange binary data in formats of your choosing. There are several examples of TCP/IP clients and servers included in the networking examples. These example VIs could be modified to run in the background with local communications via global variables. A *Data Server* VI would then act as a tunnel

for data that reappears in a *Data Client* elsewhere on the network. You can have as many connections, or *ports*, open simultaneously as you need. Several TCP/IP examples are included with LabVIEW, and you can refer to other books that cover the subject in depth (Johnson 1998; Travis 2000).

The **VI Server** includes transparent, multiplatform network access. LabVIEW itself can be the target, or particular VIs can be manipulated across the network in the same ways that we have already described.

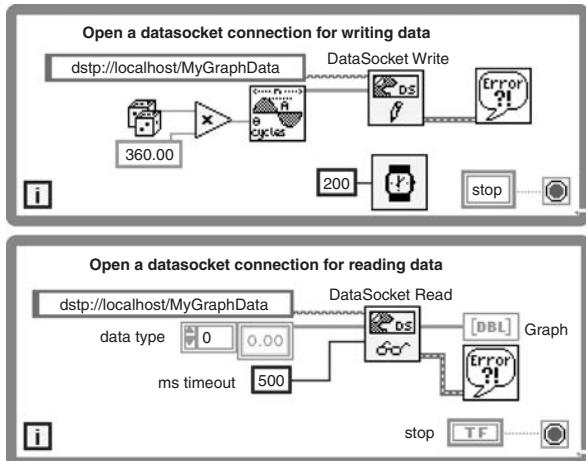
This makes it possible, for instance, to use a Call By Reference node on a Windows machine to execute a VI on a remote Macintosh running LabVIEW. An example VI shows how to do exactly that. Remember that you have to set up the VI Server access permissions in the LabVIEW Options dialog.

**DataSockets** are a great cross-platform way to transfer many kinds of data. A DataSocket connection consists of a client and a server that may or may not reside on the same physical machine. You specify the source or destination of the data through the familiar URL designator, just like you'd use in your Web browser. The native protocol for DataSocket connections is **DataSocket Transport Protocol (dstp)**.

To use this protocol, you must run a DataSocket server, an application external to LabVIEW that is only available on the Windows platform as of this writing. You can also directly access files, File Transfer Protocol (ftp) servers, and OPC servers. Here are a few example URLs that represent valid DataSocket connections:

- `dstp://servername.com/numericdata` where *numericdata* is the named tag
- `opc:\machine\National Instruments.OPCModbus\Modbus Demo Box.4:0`
- `ftp://ftp.natinst.com/datasocket/ping.wav`
- `file:\machine\mydata\ping.wav`

Figure 18.30 shows a very simple example where data is written in one While Loop and read in another. The **DataSocket Write** function includes inputs for the URL and polymorphic data. In this case we're writing a numeric array to a URL that specifies the local machine and a tag called *MyGraphData*. The **DataSocket Read** function is similar, requiring a URL and a data type specifier. Since the URLs match, the two loops in this example represent a server (writer) and a client (reader). The LabVIEW examples include a nice remote-access demonstration that reads weather data from a machine running LabVIEW at National Instruments headquarters in Austin, Texas. As you can see, DataSockets are a flexible data tunnel that requires no special programming on your part.



**Figure 18.30** Two While Loops share data via a DataSocket connection. The loops could just as well reside on different computers.

**Files as mailboxes.** An old, reliable way to exchange data between VIs over the network is to use files as mailboxes that any node can remotely fetch and read or write. For one-way transmission, files are easy to use. Just write data to a file in some predetermined format on a periodic basis, then the remote node can fetch it asynchronously, read, and decode the information. Things get tricky when you need synchronized, bidirectional communications because you must create a system of file access permission limits using flags or some other technique.

### Real-time process control databases

If you buy a commercial DCS or PC-based industrial process control package, you will find that the entire software environment is **database-driven**. For user interaction, there is invariably a program called the Database Builder that edits and compiles all the configuration information. The run-time programs are then centered around this database: I/O handlers pass values in and out to hardware, alarm handlers compare measurements with limit values, and trenders display values on the screen. LabVIEW, however, is not in itself a database-driven package. In fact, it would be much less flexible if it were centered around a predefined database structure.

A particular requirement for a database used in process control is that it must be a *real-time* database, which implies that it is memory-resident and available for thousands of transactions per second. For added versatility, the database may also be distributed among various nodes in the system. This permits any node to efficiently exchange

information with any other node on the network through a consistent message-passing scheme. A global, real-time, database requires a sophisticated client-server or peer-to-peer relationship among all nodes on the network. Another attribute of a good database is access by name. That is, you supply the tag name and attribute (such as Setpoint or Alarm High) that you wish to access, and the database looks up the name and accesses the correct items. This eliminates the headaches we've all experienced trying to keep track of array index values when data is carried in arrays. Such programming is possible in LabVIEW, but is beyond the scope of this book and beyond the abilities of all but the most experienced programmers. This is one of the areas where you may want to turn to DSC or another SCADA product.

### Simulation for validation

An effective way of testing or validating your process control system is by simulating real I/O. The obvious way is to replace each physical input with an appropriate control or calculated data source and replace each physical output with an appropriate indicator. If you've designed your application with the client-server model discussed here, simulation turns out to be quite easy. When we wrote our first well-designed client-server-based system, we didn't have any hardware for testing, so it was imperative that the software have a built-in test mode. Since the inputs all came from a PLC, the core of the input scanner VI consisted of a series of PLC input operations, and all values were collected in a single array for use by the clients. We created a boolean control that selected either *normal* or *simulation* mode and used that to control a Case structure. The *normal* frame of the Case structure contained the PLC programming, while the *simulation* frame contained a set of mathematical expressions in a big Formula Node. There was one expression for each signal, and all results were collected into an array, just like the real data. To keep things interesting, the formulas contained sine functions and random number generators.

The status of almost all output signals in a process control system are probably displayed by indicators on various panels. For simulation, then, all you must do is disable the VIs that talk to the hardware. If your outputs are handled by output servers, encapsulate the hardware output VIs in a Case structure tied to a mode control, just like we did for those simulated inputs.

If your system has closed-loop controls, where inputs respond to outputs, consider writing output data to global variables that can be read by the input simulation code. Then add some scale factors or perhaps some time-dependent response to this feedback data. The PID Control Toolkit contains a simple plant simulator that does such a trick.

There are great payoffs to built-in simulation. Most importantly, you can do a thorough job of testing your application offline. That means you can work anywhere you like, and you don't have to wait for time on the real equipment. There is a much lower risk of sending the real process out of control, losing data, or encountering some other unforeseen condition or bug. As a bonus, the simulation mode is handy for demonstration purposes, like showing the boss how crafty you are.

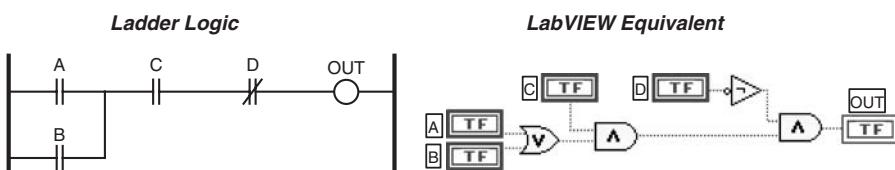
## Sequential Control

Every process has some need for sequential control in the form of interlocking or time-ordered events. We usually include manual inputs in this area—virtual switches and buttons that open valves, start motors, and the like. This is the great bastion of PLCs, but you can do an admirable job in LabVIEW without too much work. We've already discussed methods by which you read, write, and distribute data. The examples that follow fit between the input and output handlers.

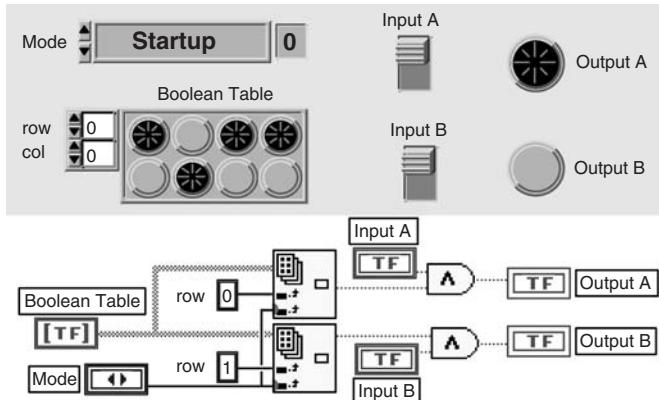
### Interlocking with logic and tables

G's boolean logic functions make ordinary interlocking simple (Figure 18.31). Boolean conditions can be derived from a variety of sources such as front panel switches, digital inputs, or comparison functions acting on analog signals. **Ladder logic**, as shown in Figure 18.31, is the most common language for PLC programming. It comes from the days of electromechanical switching and remains a useful documentation technique for logical systems.

One way to add versatility to your logical operations is to use a **boolean table**, as shown in Figure 18.32. A table is a two-dimensional array where the columns are the mode or step number and the elements in each row represent some state of the system, such as a permissive interlock. The **Mode** control selects the column through the use of a 2D **Index Array** function. The output of the Index Array function is a 1D array of booleans—a column slice from the incoming table. Another set of Index Array functions selects the individual interlock permissives.



**Figure 18.31** Simple interlock logic, comparing a ladder logic network with its LabVIEW equivalent.



**Figure 18.32** An interlock table. The 2D boolean array (a table) is sliced by an array indexer to deliver an interlock permissive vector, which then is ANDed with a series of inputs.

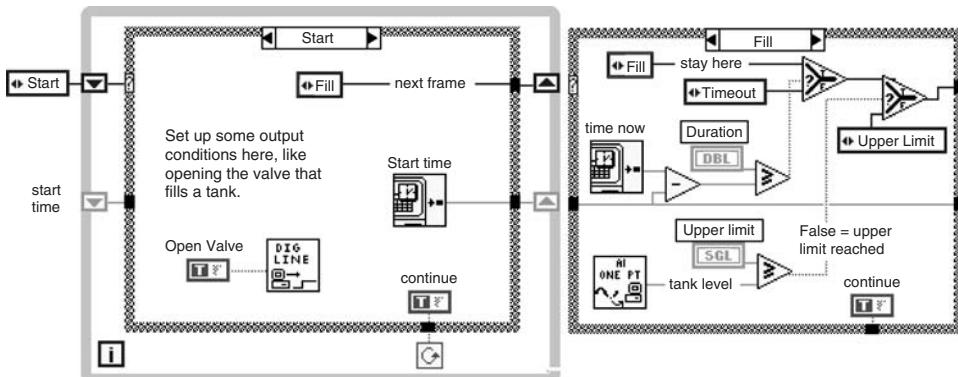
If one of the input switches is True *and* its corresponding permissive (from the table) is True, then the output is True. You can add whatever kind of logic is required and expand the table as necessary.

The **Mode** control could be a Ring control that selects between system modes such as *Startup*, *Run*, and *Maintenance*. This is an effective way to change the action of a large number of interlock chains without a great deal of wiring. Another use for the **Mode** control is to tie it to an internally generated value in your program. For instance, the mode might change in response to the time of day or the temperature in a reactor.

Tables are an efficient way to store interlock information, and they are very fast. The only bad thing about tables is that they can be hard to debug. When a table is very large and your program changes modes very often, it's not always obvious what is going on. Directly wired logic is generally easier to understand and debug, and should be used preferentially.

## State machines

The **state machine** architecture is about the most powerful LabVIEW solution for sequential control problems. A State Machine consists of a Case structure inside of a While Loop with the Case selector carried in a Shift Register. Each frame of the state machine's Case structure has the ability to transfer control to any other frame on the next iteration or to cause immediate termination of the While Loop. This allows you to perform operations in any order depending on any number of conditions—the very essence of sequential control.



**Figure 18.33** This state machine implements an operation that terminates when the tank is full or when a time limit has passed. The Timeout of the Case (not shown) takes action if a time-out occurs. The Upper Limit frame (also not shown) is activated when the upper limit for tank level is exceeded.

Figure 18.33 is an example of a tank-filling operation. The objective is to fill the tank, stopping when an upper level is reached or when a certain time has passed. In the first frame, *Start*, you open a valve (presumably by writing to an output device) that starts filling the tank. The starting time is saved in a shift register. The program then proceeds to the second frame, *Fill*, where the elapsed time and present tank level are checked. The tank level is obtained from an analog input or from a global variable, which in turn is written by an analog input handler. If the elapsed time exceeds the specified duration, the program goes to the *Timeout* frame. If the upper limit is exceeded, jump to the *Upper Limit* frame. Otherwise, keep looping on the *Fill* frame. The *Timeout* and *Upper Limit* frames may then take appropriate action to close the valve, or whatever; other operations may follow.

More activity can be managed in this same structure by looping over several frames rather than just one. The sequence could be 1-2-3-1-2-3, and so forth, with the termination conditions causing a jump to the fourth or fifth frame. Very complex looping is possible, although, like any scheme, it can be hard to debug if you're not careful. We usually put an indicator on the panel that shows which state it's in at the moment. Single-stepping then shows the execution order. If the process has to run fast, you can accumulate a history of all the states by building an array on the boundary of the While Loop for later review.

Multiple state machines can run in parallel with each managing a different task. You could have several on one diagram with a global boolean that requests all the loops to stop. This would permit you to break your control problem down into logical, related pieces that are easier to design and debug.

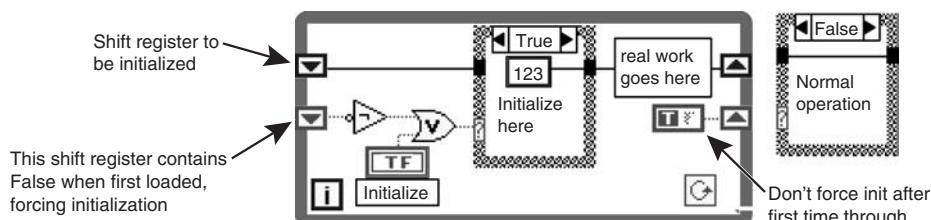
## Initialization problems

Initialization is important in all control situations and particularly so in batch processes that spend much of their time in the start-up phase. When you first load your LabVIEW program, or when you restart for some reason, the program needs to know the state of each input and output to prevent a jarring process upset. For instance, the default values for all your front panel controls may or may not be the right values to send to the output devices. A good deal of thought is necessary with regard to this initialization problem. The DSC database includes user-defined initialization for all I/O points, drastically reducing the amount of programming you'll need to do.

When your program starts, a predictable start-up sequence is necessary to avoid output transients. Begin by scanning all the inputs. It's certainly a safe operation, and you probably need some input data in order to set any outputs. Then compute and initialize any output values. If the values are stored in global variables, the problem is somewhat easier because a special initialization VI may be able to write the desired settings without going through all the control algorithms. Finally, call the output handler VI(s) to transfer the settings to the output devices. Also remember to initialize front panel controls, as discussed earlier.

Control algorithms may need initialization as well. If you use any uninitialized Shift Registers to store state information, add initialization. The method in Figure 18.34 qualifies as another Canonical VI. The technique relies on the fact that an uninitialized boolean Shift Register contains False when the VI is freshly loaded or compiled. (Remember that once the VI has been run, this is no longer the case.) The Shift Register is tested, and if it's False, some initialization logic inside the Case structure is executed. A boolean control called *Initialize* is included to permit programmatic initialization at any time, such as during restarts.

You could also send flags to your control algorithms via dedicated controls or global variables. Such a flag might cause the clearing of



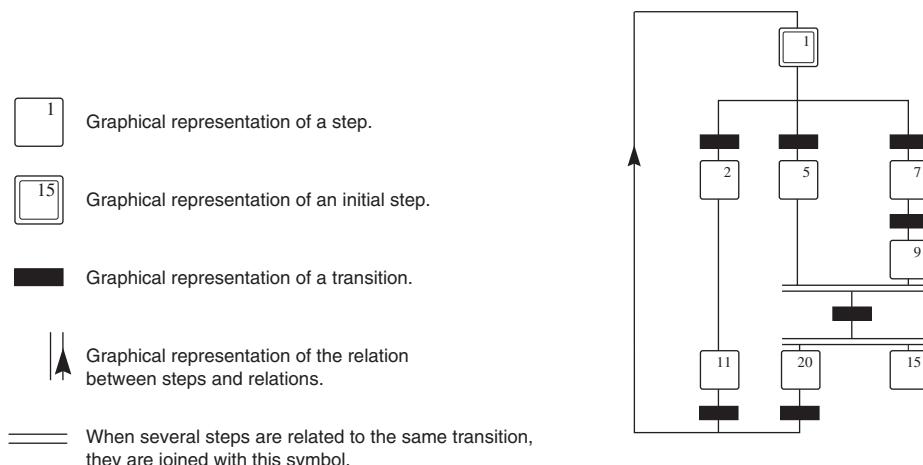
**Figure 18.34** The upper Shift Register, perhaps used for a control algorithm, is initialized by the lower Shift Register at startup or by setting the Initialize control to True.

intermediate calculations, generation of empty arrays for data storage, or cause the outputs to go to a desired state. All of these items are worth considering. Be sure to test your control system thoroughly by stopping and then restarting in various states. Start-up is the time where most process disasters occur. Of course, bad things never happen on *our* projects.

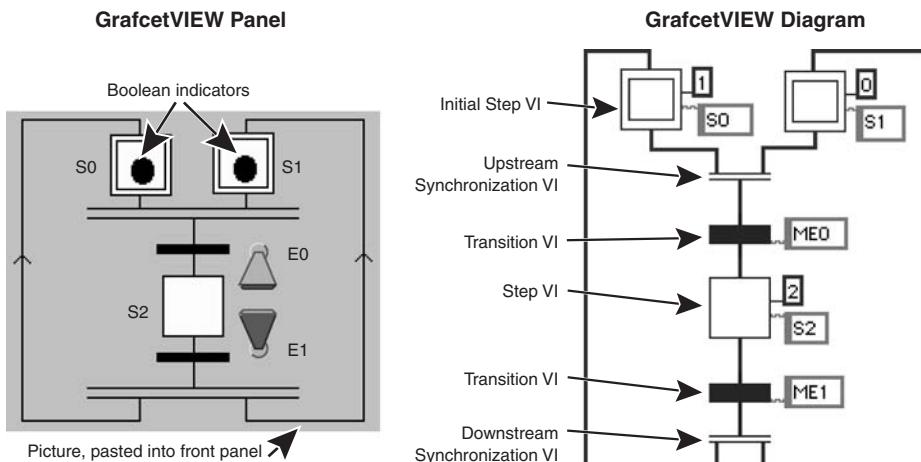
### GrafcetVIEW—a graphical process control package

There is an international standard for process control programming called **GRAFCET**. It's more popular in Europe, having originated in France, but is found occasionally in the United States and other countries. You usually need a special GRAFCET programming package, much like you need a ladder logic package to program a target machine, which is usually a PLC. The language is the forerunner of **sequential function charts (SFCs)** (IEC standard 848), which are similar to flowcharts, but optimized for process control and PLCs in particular (Figure 18.35). Emmanuel Geveaux and Francis Cottet at LISI/ENSMA in conjunction with Saphir (all in France) have developed a LabVIEW package called GrafcetVIEW that allows you to do GRAFCET programming on LabVIEW diagrams. It's a natural match because of the graphical nature of both languages.

One enhancement to LabVIEW that the authors had to make was additional synchronization through the use of **semaphores**, a classic software handshaking technique. In many control schemes, you may



**Figure 18.35** The GRAFCET language is a kind of sequential function chart. Once you understand the symbols, it's a reasonably clear way to describe a sequential operation. (*Courtesy Emmanuel Geveaux, LISI/ENSMA.*)



**Figure 18.36** This is one of the GrafsetVIEW demonstration VIs. The panel graphic was pasted in, and boolean controls and indicators were placed on top. Compare the LabVIEW diagram with actual GRAFCET programming in Figure 18.35.

have parallel tasks that execute continuously and independently, but which occasionally need to force one another to pause to guarantee sequential timing during certain operations. The GrafsetVIEW solution is a set of subVIs that manages semaphores. It may also be possible to implement the solution via Occurrences, but their semaphore trick seems to be immune to some of the difficult starting and ending occurrence generation situations.

GrafsetVIEW requires only a few subVIs to mimic the GRAFCET language, and they are wired together in a manner that matches the drawings. Figure 18.36 shows the panel and diagram of one of the GrafsetVIEW example programs. The most remarkable feature of this diagram is that it apparently implements **cycles**, or closed loops, in LabVIEW. As you may know, cycles of any type are illegal; just try wiring the output of a function or VI to its input. That's why G is properly called an **acyclic dataflow** programming language. So how did they do it? If you inspect the GrafsetVIEW subVI terminals, the wiring really does flow from inputs to outputs in the usual, legal manner, but the routing makes it *seem* cyclic. Amazing!

Once your GRAFCET logic diagram has been built into a subVI, you combine it with some run-time VIs that initialize the digital I/O hardware and others that actually execute the logic with appropriate synchronization. GrafsetVIEW uses a series of global variables to pass information about the logic to the run-time engine VIs. The final diagrams are very concise. The package also makes it easy to simulate the

behavior of an application by replacing actual I/O points with boolean controls and indicators. Such a validation technique is valuable in any control implementation.

Even if you're not planning to run your system with GrafctVIEW, it makes a good learning and demonstration tool for the language. For more information on GRAFCET, visit [www.tecatlant.fr](http://www.tecatlant.fr).

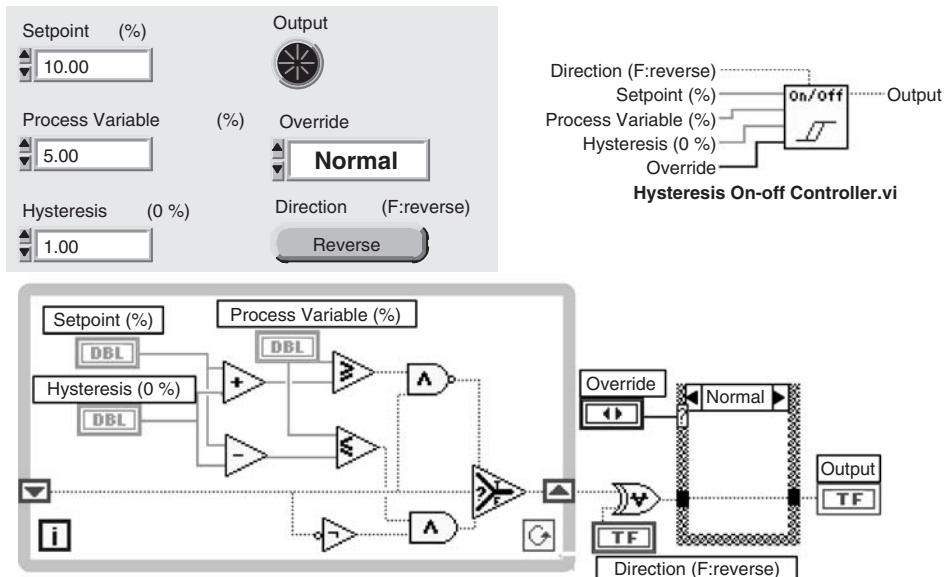
## Continuous Control

**Continuous control** generally implies that a steady-state condition is reached in a process and that feedback stabilizes the operation over some prolonged period of time. Single-loop controllers, PLCs, and other programmable devices are well suited to continuous control tasks, or you can program LabVIEW to perform the low-level feedback algorithms and have it orchestrate the overall control scheme. LabVIEW has some advantages, particularly in experimental systems, because it's so easy to reconfigure. Also, you can handle tricky linearizations and complex situations that are quite difficult with dedicated controllers. Not to mention the free user interface.

Most processes use some form of the PID algorithm as the basis for feedback control. Gary wrote the original PID VIs in the LabVIEW **PID Control Toolkit** with the goal that they should be easy to apply and easy to modify. Every control engineer has personal preferences as to which flavor of PID algorithm should be used in any particular situation. You can easily rewrite the supplied PID functions to incorporate your favorite algorithm. Just because he programmed this particular set (which he personally trusts) doesn't mean it's always the best for every application. The three algorithms in the package are

- **PID**—an interacting positional PID algorithm with derivative action on the process variable only
- **PID (Gain Schedule)**—similar to the PID, but adds a tuning schedule that helps to optimize response in nonlinear processes
- **PID with Autotuning**—adds an autotuning wizard feature that guides you through the loop tuning process

There is also a **Lead/Lag** VI that is useful for more advanced control strategies, such as feedforward control. **Lead** refers to the phase shift associated with a single-pole high-pass filter or differentiator, while **lag** refers to a low-pass filter or integrator. The Lead/Lag VI is also useful in simulations where you need to emulate the time-domain response of a first-order system. We won't elaborate on the contents of the whole package here. Just order the toolkit from National Instruments and read the manual.

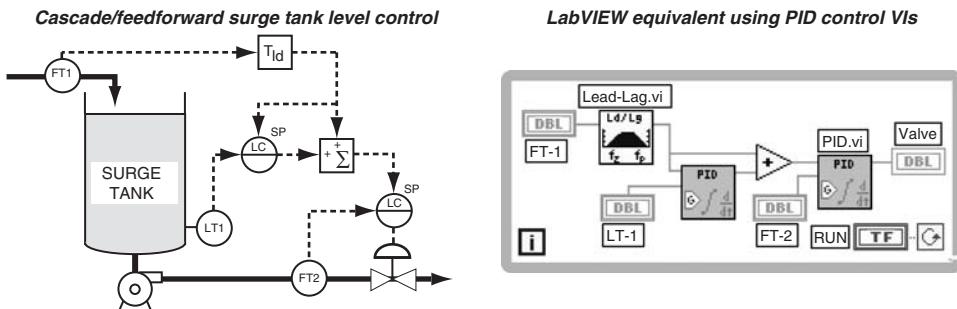


**Figure 18.37** An on/off controller with hysteresis, which functions much like a thermostat. Add it to your process control library.

You can use the math and logic functions in LabVIEW to implement almost any other continuous control technique. For instance, on/off or bang-bang control operates much like the thermostat in your refrigerator. To program a simple on/off control scheme, all you need to do is compare the setpoint with the process variable and drive the output accordingly. A comparison function does the trick, although you might want to add hysteresis to prevent short-cycling of the output, as we did in the **Hysteresis On/Off Controller VI** in Figure 18.37. This control algorithm is similar to the ones in the PID Control Toolkit in that it uses an uninitialized shift register to remember the previous state of the output.

### Designing a control strategy

The first step in designing a continuous control strategy is to sketch a flowchart of your process, showing control elements (i.e., valves) and measurements. Then add feedback controllers and any other required computations, as textbooks on the subject of process control recommend. The finished product will probably look like some kind of a P&ID diagram. Your goal is to translate this diagram into a working LabVIEW system. The question is whether to have LabVIEW do the real-time feedback control loop calculations or to have an external smart controller



**Figure 18.38** With the PID functions, you can map a control strategy from a textbook diagram to a LabVIEW diagram.

do the job. As we've already discussed, the choice depends on performance and reliability requirements as well as personal preference.

If you decide to have LabVIEW do the processing, translate the flowchart into a LabVIEW block diagram using the PID control VIs with the math and logic functions of LabVIEW. An example of a translated control scheme is shown in Figure 18.38. The only elements missing from this simplified program are the loop tuning parameters and auto/manual switching.

If you use an external controller, the program is even simpler. All you will have to do is write settings such as setpoints and tuning parameters and read the status of various measurements from the controller.

You can supplement the functionality of the external controllers with math and logic in your LabVIEW program. In all cases, there will be the usual requirements for interlocking and initialization.

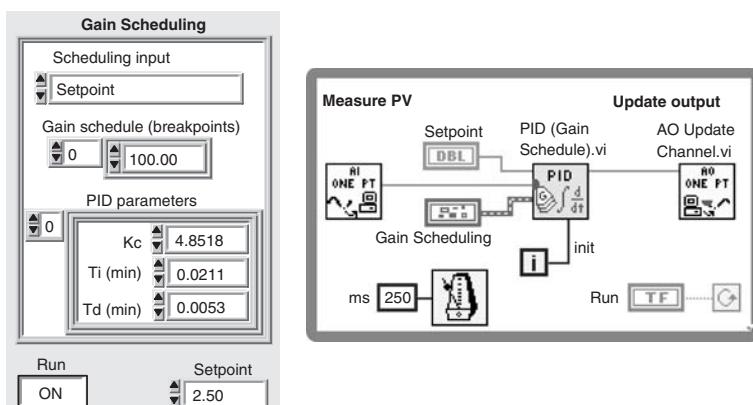
Your choice of control topology must complement the process, or the resulting performance may suffer from instability, overshoot, or long-term error, and it may be very difficult to tune or optimize. This is where training in process control strategies, books, and practical experience come into play. It's better to choose the right approach, if possible, rather than force-fitting a generic controller into a difficult process situation. Here is one very common example, taken from an exchange Gary had with someone on the info-labview mailgroup.

*Question:* I am using the PID algorithm to control temperature. I would like to make a step change in controlled temperature as sharp as possible without overshoot and ripples.

Seems like a reasonable request, doesn't it? Well, it turns out that control engineers have been struggling with this exact problem for the last 60 years with mixed results and a very wide range of possible solutions, depending upon the exact character of the process, the available

control equipment, and the sophistication of the engineer. We can list a few solutions that you might want to look into.

1. Use **feedforward** control techniques. Basically, this is where you characterize your process (temperature versus controller output, both static and dynamic response) and force the output to move in the right direction with the proper dynamics to compensate for the response of the process. Feedforward is quite safe as far as instability is concerned. The Lead/Lag VI was included in the PID Control Toolkit for such control schemes. You can read about it in several books (Corripi 1990; McMillan 1995; Shinskey 1988). We would use this technique as a first choice. Note that it's not available in single-loop controllers, but LabVIEW can do it.
2. Create a **PID tuning schedule**. Your process gain and temporal response vary with absolute temperature, which implies that your PID tuning must be different at every temperature. Process gain is the temperature change divided by the controller output change. Typically, the process gain is much lower at high temperatures, leading to sluggish response. Ideally, you create a mathematical model of your process either theoretically or by gathering empirical data. The model is then loaded into a simulation package (such as MATLAB's SimuLink), and you derive optimal control parameters from that. For simpletons like us, we generally perform an open-loop step response measurement at several temperatures, as described in the toolkit manual, and create a table of tuning parameters. Then we use the **PID (Gain Schedule) VI**, fed by a cluster containing those PID parameters. Figure 18.39 shows what it looks like. The tuning schedule



**Figure 18.39** This is one way to implement a PID tuning schedule. Properly tuned, it yields good control performance over a wide range of operating conditions.

can evaluate break points based on the process variable, setpoint, controller output, or an extra numeric input called the *gain scheduling variable*. That pretty much covers any situation you'll encounter. In some processes, you can get away with constant values for the integral and derivative terms while varying only the proportional gain.

3. Another technique related to tuning schedules is sometimes called **approach control gain adjustment** or *start-up management* and is included with some single-loop controllers. Basically, you reduce the controller's gain as the error (setpoint minus process variable) decreases. This can reduce overshoots (sometimes). It generally fails with processes that have a long dead time.
4. Ramp the setpoint. Many controllers will nicely track a ramped setpoint without the severe upset normally seen with large step changes in setpoint. You can add a profile to the ramp to further improve the response. Typically, the ramp's rate of change can be larger at the outset, decreasing as it nears the final value.
5. If you want to use a simple PID algorithm, use as much proportional gain as the process can stand without oscillation. The proportional term is your friend. Also be sure to use derivative, which is a stabilizing feedback term. In fact, you can make a PD controller, thus avoiding the reset (integral) term which encourages overshoot. The trouble is, there will be some long-term error, which you can cancel by adding a dc bias to the controller output. The bias is determined experimentally. For one experiment, we generated lots of small steps in temperature, recording the process response, then determined a bias schedule, which can be a piecewise linear fit. Resulting error is pretty low for an invariant process. An alternative is a control scheme that switches modes, from PD at start-up to PID at steady state. Again, a tuning schedule can perform this trick. Look up **batch controllers** in Shinskey (1988). Batch systems cannot tolerate overshoot.
6. Consider **fuzzy logic**, **predictive**, or **model-based** controllers. These are advanced control schemes beyond the scope of this book, but exceptional performance is possible, with enough effort.

**Automatic parameter tuning** is available in the PID Toolkit and from VI Engineering. Its Automatic PID Tuner can help you select optimum tuning parameters using a variety of well-known analysis techniques. Note that these tools are not for continuous, online use, so they won't automatically solve your *dynamic* problems.

**Scaling input and output values.** All the functions in the PID Control Toolkit have a cluster input that allows you to assign setpoint,

process variable, and output ranges. You supply a *low* and a *high* value, and the controller algorithm will normalize that to a zero to 100 percent range or **span**, which is customarily used for PID calculations.

Here is an example. Assume that you use a multifunction board analog input to acquire a temperature signal with a range of interest spanning 0 to 250°C. You should set the *sp low* value to zero, and the *sp high* value to 250 in the PID option cluster. That way, both the setpoint and process variable will cover the same range. On the output side, perhaps you want to cover a range of 0 to 10 V. You should assign *out low* to zero and *out high* to 10. That will keep the analog output values within a sensible range.

Once everything is scaled, the PID tuning parameters will begin to make sense. For instance, say that you have a temperature transmitter scaled from -100 to +1200°C. Its span is 1300°C. If the controller has a *proportional band* of 10 percent, it means that a measurement error of 130°C relative to the setpoint is just enough to drive the controller output to saturation, if you have scaled your setpoint in a similar manner. In the PID tuning parameters, they call for a *proportional gain* value, which is equal to 100 divided by the proportional band, in percent. So for this example, the proportional gain would be 10.

**PID tricks and tips.** All of the PID VIs contain a four-sample FIR low-pass filter on the process variable. It's there to reduce high-frequency noise, which is generally a good thing. However, it also induces some additional phase shift. This extra lag must be included in any process simulation. Gary sometimes goes into the VI and rips out the filter subVI, and does his own filtering as part of the data acquisition process.

Nonlinear integral action is used to help reduce integrator windup and the associated overshoot in many processes. The error term that is applied to the integrator is scaled according to the following expression:

$$\text{Error} = \frac{\text{error}}{1 + 10(\text{error}^2/\text{span}^2)}$$

where *error* is the setpoint minus the process variable and *span* is the controller's setpoint span. Thus, when the deviation or error is very large, such as during process start-up, the integral action is reduced to about 10 percent of its normal value.

Because derivative action is effectively a high-pass filter, high-frequency noise is emphasized, leading many users to abandon derivative action altogether. To combat such problems, *derivative limiting* is sometimes used. It's basically a low-pass filter that begins acting at high frequencies to cancel the more extreme effects of the derivative. In the PID Toolkit, there is no implicit limit on the derivative action.

However, if you leave the process variable filter in place, there's a good chance that it will serve as a nice alternative.

**Timing and performance limitations.** Always be wary of timing limitations and system performance when you are doing continuous control. DCS manufacturers rate their I/O controllers in terms of *loops per second*, referring to how many PID calculations can be performed in one second under average conditions. If the system software is well written, adding more loops will cause a gradual degradation of performance rather than an outright failure of any kind. It's much better to have a loop running 20 percent slow than not at all. Also, there should be no way to accidentally upset the steady operation of a continuous control algorithm. Real processes have plenty of unpredictable features without help from cantankerous software controls.

According to control theory, a sampled control system needs to run about 10 times faster than the fastest time constant in the plant under control. For instance, a temperature control loop is probably quite slow—a time constant of 60 s is common in a small system. In this case, a cycle time of about 6 s is sufficient. Faster cycling offers little or no improvement in performance. In fact, running all your control VIs too fast degrades the overall response time of your LabVIEW application. If you use the timing functions available in LabVIEW to regulate execution of a feedback algorithm, be aware of the actual precision available on your computer—typically 1 ms. Therefore, the fastest practical loop cycle times are on the order of 10 ms (100 Hz) for most LabVIEW systems. To go faster, you must obtain better timing information (from a hardware timer) or run the algorithm on a LabVIEW RT system or use another external device with suitable performance. By the way, most industrial single-loop controllers cycle no faster than about 5 Hz (200 ms).

Here is an example of how timing accuracy can affect a control algorithm. A PID algorithm has two time-dependent terms, the *integral* and *derivative* responses. When the algorithm is called, the amount of time since the last execution, *t*, is used in the calculation. If *t* is in error, then the response of the algorithm may also be in error. The error magnitude depends on the tuning parameters of the PID as well as the state of the process. If the process is in steady state, then the time-dependent terms are zero anyway, and the timing error does not matter. But during a process upset, the response to timing errors can be very hard to predict. For best results, you had better make sure that your control loops run with a steady rhythm and at a sufficiently high speed.

The PID Control Toolkit supports either *internal* or *external* timing. Internal timing uses LabVIEW timing functions with the resolution limits previously mentioned. The advantage of this method is

that the PID functions keep track of the elapsed time between each execution. External timing requires you to supply the actual cycle time (in seconds) to the PID function VI. If you are using the DAQ library, the actual scan period for an acquisition operation is returned by the **Waveform Scan** VI, for instance, and the value is very precise. Each PID VI has an optional input called **dt**. If **dt** is set to a value less than or equal to zero seconds (the default), internal timing is used. Positive values are taken as gospel by the PID algorithm.

The DAQ analog I/O example **Analog IO Control Loop (hw timed)** is an example of a data acquisition operation where you can place a PID loop on one of the input channels driving an analog output. The trick is to wire **actual scan rate** (in scans per second) from the DAQ VI, **AI Start**, through a reciprocal function to the PID VI. This provides the PID algorithm with an accurate time interval calibration. The data acquisition board precisely times each data scan so you can be assured that the While Loop runs at the specified rate. This example should run reliably and accurately at nearly 1 kHz, including the display.

## Trending

The process control industry calls graphs and charts **trend displays**. They are further broken down into **real-time trends** and **historical trends**, depending on the timeliness of the data displayed. Exactly where the transition occurs, nobody agrees. Typically, a historical trend displays data quite a long time into the past for a process that runs continuously. A real-time trend is updated frequently and only displays a fairly recent time history. Naturally, you can blur this distinction to any degree through crafty programming. Historical trending also implies archival storage of data on disk for later review while real-time trending may not use disk files.

### Real-time trends

The obvious way to display a real-time trend is to use a **Waveform Chart** indicator. The first problem you will encounter with chart indicators is that historical data is displayed only if the panel containing the chart is showing at all times. As soon as the panel is closed, the old data is gone. If the chart is updating slowly, it could take quite a long time before the operator sees a reasonable historical record. A solution is to write a program that stores the historical data in arrays and then write the historical data to the chart with the **Chart History** item in a Property node. You should take advantage of strip charts whenever possible because they are simple and efficient and require no programming on your part.

Consider using an **XY Graph** as a real-time trend display for extra versatility. You maintain arrays that contain some number of recent measurements and another array containing accurate timestamps.

Then it's a simple matter of graphing the data versus the timestamps whenever a real-time trend needs to be displayed. If the arrays are stored in global variables, any VI can read and display data from any channel at any time.

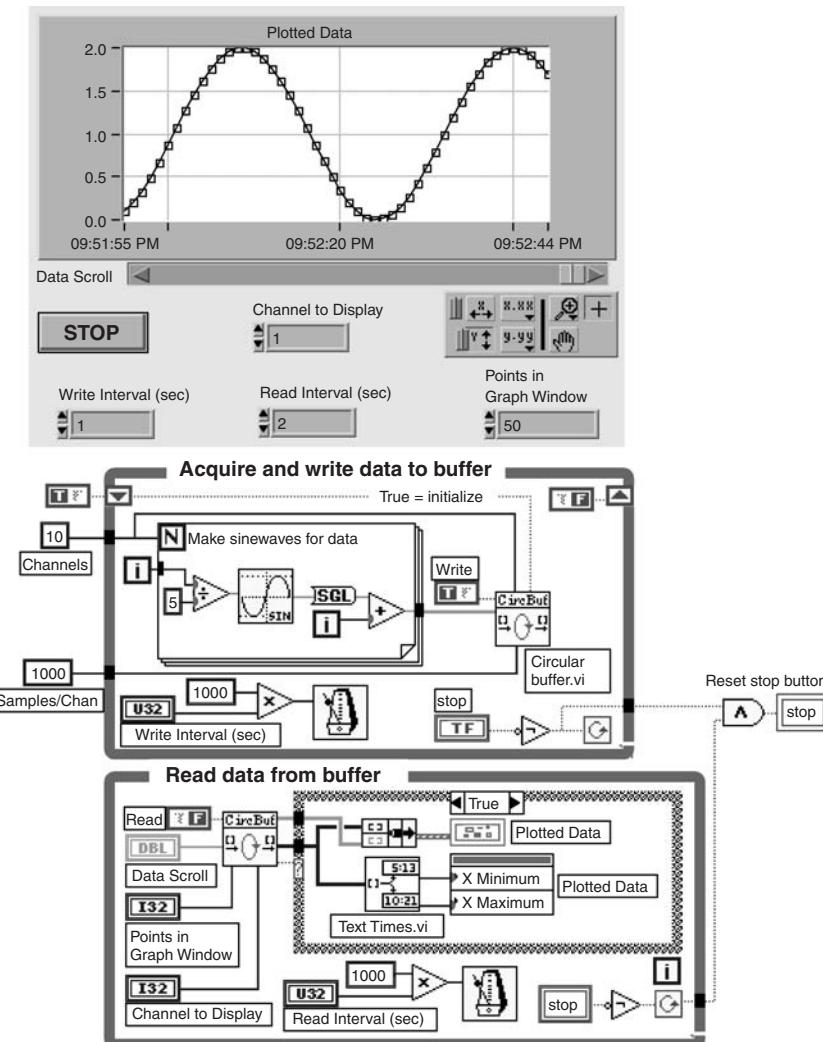
An important concern with this technique is memory management. If you just keep appending samples to your trending arrays, you will ultimately run out of memory. Also, the graphs will gradually become sluggish because of the large numbers of plot points. Some kind of length-limited array or a data compression technique is required for long-term operation.

A sophisticated method of trend data management uses a **circular buffer**. A *circular buffer* is an array that you program to act as if it is a continuous ring of data in memory rather than a straight line of finite length. Because the array never has to change size or be relocated, no real-time memory management is required and the performance is very good. The amount of data you can store is limited only by available memory. Unfortunately, the programming is rather complex: You must keep track of where the oldest and newest data are in the buffer and figure out a way to map the (physical) linear array into a (virtual) circular one.

The **Circular Buffer VI** has some interesting features that make it more useful than the regular strip charts. First, you preallocate all the required memory by setting the desired number of channels and samples per channel. Second, you periodically call the VI, supplying an array that contains the measurements for each channel. These are written to the next location in the circular buffer. At the same time, timestamps are recorded in a separate buffer. The timestamps are in epoch seconds, where zero is 1-Jan-1904. They are internally managed so you don't even have to supply a time measurement.

The interesting part comes when you want to read the data. The VI returns an array of data for any one channel and an array of timestamps suitable for *xy* plotting. The number of samples is adjustable, as is the start time of the returned array. That way, you can scroll back through time, looking at a window of data rather than trying to zoom in by using the graph controls. It's much faster, too, because the graph only has to display a limited number of points. If the buffer is sized for 10,000 samples, you might be hesitant to attempt to display them all at once.

Figure 18.40 shows the panel and diagram of an example VI that uses the Circular Buffer VI for real-time trending. Two independent While Loops store and retrieve data. Because the Circular Buffer VI is a kind of global variable, you can call it in read or write mode any place,



**Figure 18.40** An example of the Circular Buffer real-time trending subVI in action. Parallel While Loops are used, one to write sinusoidal data for 10 channels, and the other to read a specified range from a channel for display. Note that the graphed data can be scrolled back in time.

any time. The data written here is a set of sine waves, one for each of the 10 channels defined. The buffer is initialized to contain 1000 samples per channel in this example. Every two seconds, a fresh set of values for all 10 channels is written.

The **Data Scroll** slider allows you to display a range of data from the past without disturbing the data recording operation. The **Points**

**in Graph Window** control lets you display a selected number of data points on the graph. To properly display time, we used the graph's X Scale Formatting menu to set the scale to time/date. On the diagram, a Property node sets the *x* axis minimum and maximum values. This technique guarantees correct interpretation of timing regardless of actual sampling intervals because a timestamp is stored with each sample, and data is always plotted versus timestamps.

Memory usage and performance are very good. There is a noticeably long delay when you call Circular Buffer with its initialization boolean set to True. At that time, it allocates memory for the entire buffer, which can be quite large. The actual amount of memory used for the buffer itself is calculated from

$$\text{Bytes} = (N + 1) \times M \times 8$$

where *N* is the number of channels and *M* is the number of samples per buffer. For instance, 10 channels and 10,000 samples per channel requires 880,000 bytes of memory. (The 1 in the formula accounts for the hidden timestamp buffer.) Only one duplication of the data array is required, and that is also static memory usage, so the actual memory usage is twice the result from the formula. When you write data, the **Replace Array Element** function is used, so no reallocation of memory is required. When reading, the amount of memory required is proportional to the number of data points to be displayed, a number significantly smaller than the size of the buffer for practical usage.

Thanks go to Marty Vasey for his original work on this memory-based circular buffer concept. He designed it out of necessity: A customer was threatening bodily harm if they couldn't scroll back through their real-time data and do it *fast*. This program does the trick.

## Historical trends

Memory-resident data is fine for real-time trending where speed is your major objective. But long-term historical data needs to reside on disk both because you want a permanent record and because disks generally have more room. You should begin by making an estimate of the space required for historical data in your application. Consider the number of channels, recording rates, and how far back in time the records need to extend. Also, the data format and content will make a big difference in volume. Finally, you need to decide what means will be used to access the data. Let's look at some of your options.

All of the basic file formats discussed in Chapter 7, "Files,"—datalogs, ASCII text, and proprietary binary formats—are generally applicable to historical trending. ASCII text has a distinct speed *disadvantage*, however, and is probably not suited to high-performance trending

where you want to read large blocks of data from files for periodic redisplay. Surprisingly, many commercial PC-based process control applications do exactly that, and their plotting speed suffers accordingly. Do you want to wait several minutes to read a few thousand data points? Then don't use text files. On the other hand, it's nice being able to directly open a historical trending file with your favorite spreadsheet, so text files are certainly worth considering when performance is not too demanding.

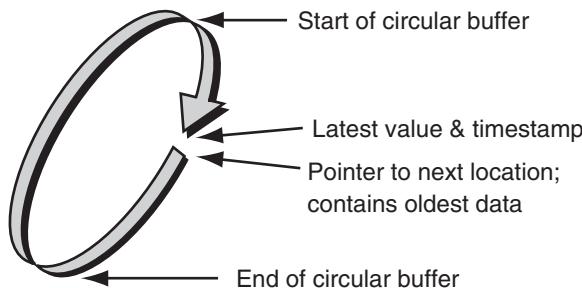
LabVIEW datalog files are a better choice for random access historical trending applications because they are fast, compact, and easy to program. However, you must remember that only LabVIEW can read such a format, unless you write custom code for the foreign application or a LabVIEW translator program that writes a more common file format for export purposes.

**The HIST package.** A custom binary file format is the optimum solution for historical trending. By using a more sophisticated storage algorithm such as a circular buffer or linked list on disk, you can directly access data from any single channel over any time range. Gary wrote such a package, called **HIST**, which he used to sell commercially.

The HIST package actually includes two versions: Fast HIST and Standard HIST. Fast HIST is based on the real-time circular buffer that we've just discussed, but adds the ability to record data to disk in either binary or tab-delimited text format. This approach works well for many data acquisition and process control packages. Based on a single, integrated subVI, Fast HIST has very low overhead and is capable of recording 100 channels at about 80 Hz to both the memory-based circular buffer and to disk. Utility VIs are included for reading data, which is particularly important for the higher-performance binary format files.

Standard HIST is based on a suite of VIs that set up the files, store data, and read data using circular buffers *on disk* rather than in memory. At start-up time, you determine how many channels are to be trended and how many samples are to be saved in the circular buffers on disk. Sampling rates are variable on a per channel basis. The use of circular buffers permits infinite record lengths without worry of overflowing your disk. However, this also means that old data will eventually be overwritten. Through judicious choice of buffer length and sampling parameters and periodic dumping of data to other files, you can effectively trend forever. (See Figure 18.41.)

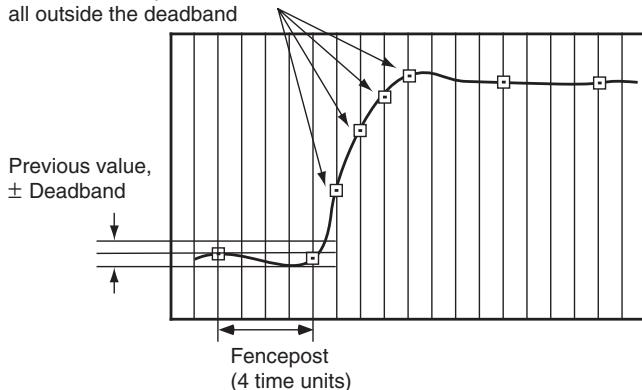
Data compression is another novel feature of Standard HIST. There are two compression parameters—**fencepost** and **deadband**—for each channel (Figure 18.42). *Fencepost* is a guaranteed maximum update time. *Deadband* is the amount by which a channel's value



**Figure 18.41** Representation of a circular buffer on disc. This buffer contains the values and timestamps from a single channel. Other channels would be located before and after this one. The program must keep track of pointers to the start, end, and next locations in the file.

must change before it is updated on disk. The combination of these two parameters is very flexible. If deadband is zero, then the channel will be updated each time the **Store HIST Data VI** is called (as fast as once per second). If deadband is a very large number, then the channel will be updated every time the fencepost period passes. If you choose a moderate deadband value, then transients will be stored with high fidelity while steady-state conditions will be trended with a minimum frequency to preserve buffer space. Each time you store data, you update each channel's fencepost and deadband (data compression parameters), giving real-time control over which channels are being

These four samples were all outside the deadband



**Figure 18.42** Illustration of the action of deadband-fencepost data compression. Values are stored at guaranteed intervals determined by the fencepost setting, which is four time units in this example. Also, a value is stored when it deviates from the last stored value by greater than  $\pm$ deadband (measured in engineering units).

trended at what rates. Similar data compression techniques are found in DSC and in many other process control packages.

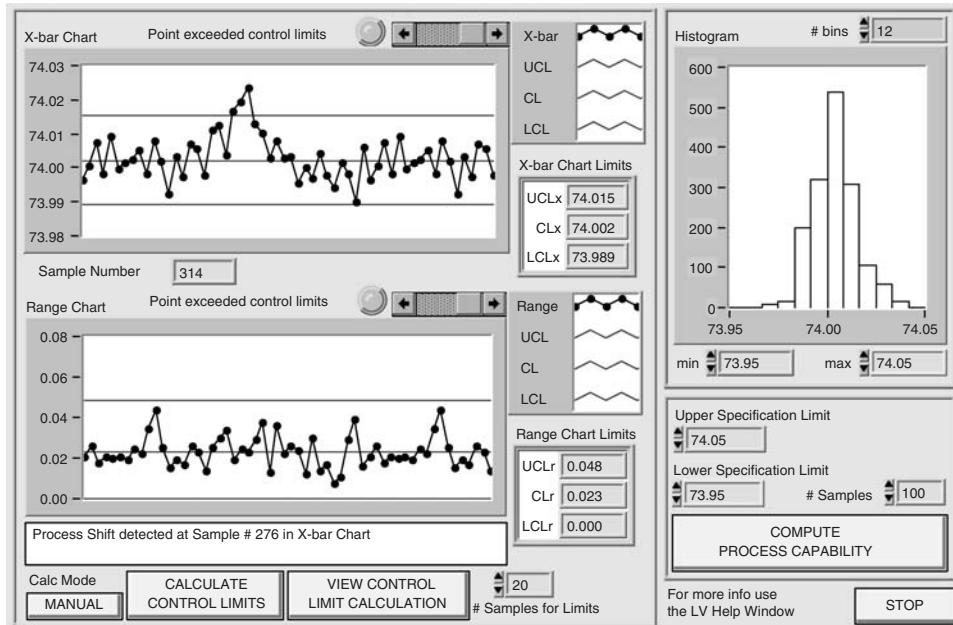
Standard HIST includes all the necessary file services, such as file creation; in addition, it allows you to restart historical trending with an existing set of files, picking up right where you left off. This is possible because the indexing information in the circular buffer data file is kept in a separate file. Also, you can run up to four separate HIST file sets simultaneously. That permits you to break up your data as you see fit. The data files are in a special binary format which makes reading the data in other programs impossible without writing special programs. For this reason, a VI called **HIST Data to Text File** is included to do a translation to tab-delimited text within LabVIEW. You could also write your own translation VIs since the data from every channel is readily available in ordinary LabVIEW arrays.

### Statistical process control (SPC)

What do you do with tons of data you've collected from monitoring your process or test system? In manufacturing situations, **statistical process control (SPC)**, also known as **statistical quality control (SQC)**, techniques are commonly used to emphasize the range over which the process is in control and what the process is capable of producing. Once a process is in control, it can be improved. Statistical techniques are used to monitor the mean and variability of a process. There is always some random variability in a process, but there may also be other non-random causes (i.e., systematic errors) present which must be identified and corrected.

SPC techniques aid in identifying whether or not special causes are present so that the process is corrected only when necessary. A process parameter is usually plotted against *control limits*. If the process parameter exceeds these control limits, there is a very high probability that the process is not in control, which means that there is some special cause present in the process. This special cause could be a new piece of equipment, a new untrained operator, or sundry other problems. Once a process is in control, SPC techniques can predict the yield or defect rate (in parts per million, for example) of the process. There are also techniques for designing experiments in order to improve the process capability. SPC techniques are pretty popular these days with everybody trying to meet ISO 9000 quality criteria. Some good references for applying SPC techniques are listed at the end of this chapter (Montgomery 1992; Wheeler and Chambers 1992).

The **Statistical Process Control Tools**, available from National Instruments as part of the Enterprise Connectivity Toolset, makes a great starting point for any SPC application (Figure 18.43). It comes

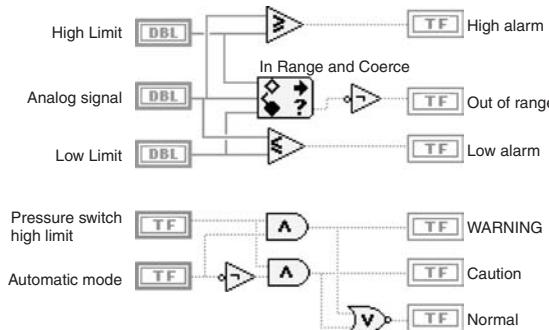


**Figure 18.43** The SPC Toolkit demo VI displays several of the built-in chart types and statistical analysis techniques included in the package.

with a comprehensive set of VIs and examples and a thorough manual. Common presentation techniques for SPC are *control charts*, *Pareto analysis*, *histograms*, and other statistical calculations. To make your life easier, the toolkit includes a set of graph and table controls that are customized for these standard presentations. Special VIs compute process statistics and prepare data for display. Most of the VIs expect arrays of data for processing, so you can either load historical data from disk or maintain recent data in memory, perhaps in a circular buffer. Application is straightforward, but be warned that the world of SPC can be daunting to the newcomer. Don't expect magical improvements in your process unless you study the references or get clear instructions from an expert.

## Alarms

A vital function of any process control system is to alert the operator when important parameters have deviated outside specified limits. Any signal can be a source of an **alarm** condition, whether it's a measurement from a transducer, a calculated value such as an SQC control limit, or the status of an output. Alarms are generally classified



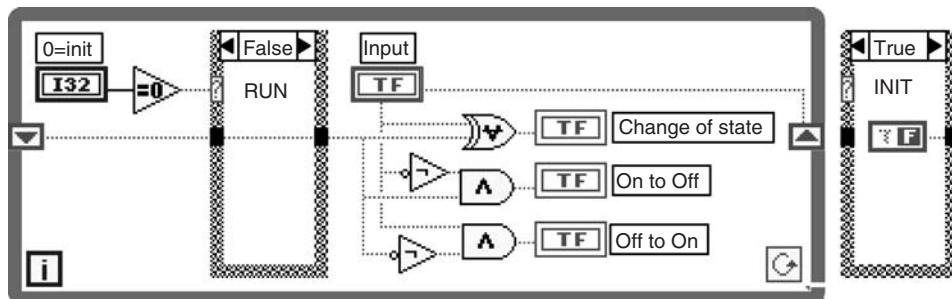
**Figure 18.44** Using comparison functions to detect alarm conditions on analog signals (top). Digital signals are combined with boolean logic to generate alarm states (bottom).

by severity, such as *informative*, *caution*, or *warning*, and there are a variety of audiovisual alarm presentation methods available.

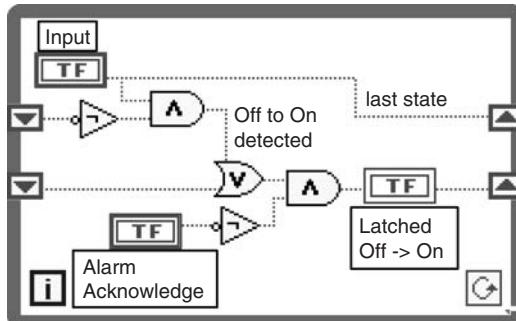
You should start by listing the signals and conditions in your process that require some form of alarm response and determining the appropriate responses. For analog signals and computed values, use the **comparison** functions (Figure 18.44, top) to generate boolean alarm flags. Digital signals, such as contact closure inputs, are combined with the logic functions, such as AND, OR, and NOT, to generate flags (Figure 18.44, bottom). Naturally, you can combine these techniques as required.

**Change-of-state detection** is another important function that you may need for alarm generation. This enables you to react to a signal that is generally constant but occasionally undergoes a sudden deviation.

To detect a change of state, the VI must remember the previous state of the signal, which implies the use of a Shift Register as shown in Figure 18.45.



**Figure 18.45** A change of state can be detected by comparing a boolean's present value with its previous value stored in a Shift Register. Three different comparisons are shown here. This would make a reasonable subVI for general use.



**Figure 18.46** This example latches an alarm state until the Alarm Acknowledge input is set to True.

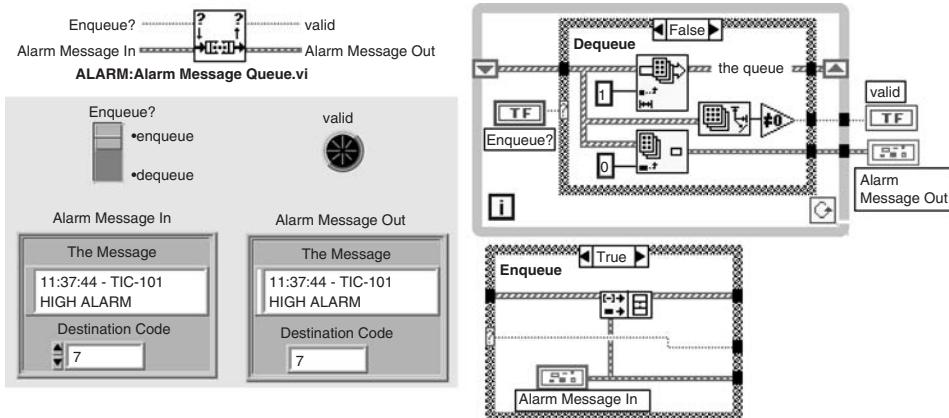
Whenever the input changes state, the appropriate boolean indicators will be set to True. Because the logic functions are polymorphic, you can change all the boolean controls and indicators to arrays of bools without rewiring these last two VIs (Figs. 18.44 and 18.45). This allows you to process several channels at once. If you *do* modify this example to use arrays, the initial array inside the Case must contain the desired number of elements.

An extension of this technique employs **latching**, used when you need the boolean indicator to remain in the alarm state until a separate reset signal clears the alarm. The reset function is called **alarm acknowledge** in process control terminology. In Figure 18.46, the upper Shift Register detects an off-to-on change of state, as in the previous example. The lower Shift Register is set to True when a change of state occurs, and stays that way until **Alarm Acknowledge** is True. Again, the inputs and outputs could be arrays.

### Using an alarm handler

Detecting alarms is the easy part. But there is a potential data distribution problem just like we encountered with input and output signals. If your system is fairly complex, alarms may be generated by several VIs with a need for display elsewhere. An **alarm handler** adds a great deal of versatility to your process control system, much as the I/O handlers do for analog and digital signals. It acts as a centralized clearinghouse for alarm messages and status information.

**Global Alarm Queue VI.** One way to handle distributed alarm generation is to store alarm messages in a **global queue**. The approach is similar to the one we used with an output handler VI. Multiple VIs can deposit alarm messages in the queue for later retrieval by the alarm handler.

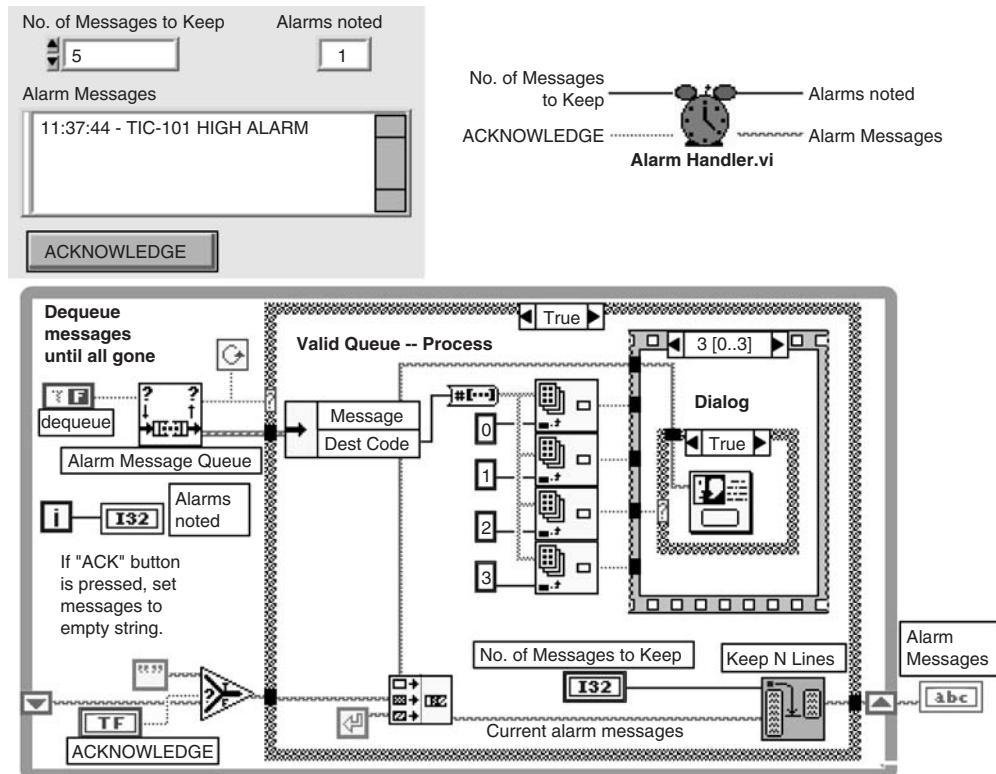


**Figure 18.47** A global queue stores alarm messages. A message consists of a cluster containing a string and a numeric. In Enqueue mode, elements are appended to the array carried in the Shift Register. In Dequeue mode, elements are removed one at a time.

A queue guarantees that the oldest messages are handled first and that there is no chance that a message will be missed because of a timing error. The alarm message queue in Figure 18.47 was lifted directly from the Global Queue utility VI that was supplied with older versions of LabVIEW. All we did is change the original numeric inputs and outputs to clusters. The clusters contain a message string and a numeric that tells the alarm handler what to do with the message. You could add other items as necessary. This is an unbounded queue which grows without limits.

Alarms can be generated anywhere in your VI hierarchy, but the I/O handlers may be the best places to do so because they have full access to most of the signals that you would want to alarm. You can combine the output of an alarm detector with information from your configuration database to produce a suitable alarm message based on the alarm condition. For instance, one channel may only need a high-limit alarm while another needs both high- and low-limit alarms. And the contents of the message will probably be different for each case. All of these dependencies can be carried along in the configuration. Once you have formatted the message, deposit it in the global alarm queue.

**Alarm Handler VI.** Once the alarm messages are queued up, an alarm handler can dequeue them asynchronously and then report or distribute them as required. The alarm handler in Figure 18.48 performs two such actions: (1) It reports each alarm by one of four means as determined by the bits set in the **Destination Code** number that's part of the message cluster; and (2) it appends message strings to a string indicator,



**Figure 18.48** An alarm handler VI. It reads messages from the global alarm queue and reports them according to the bits set in the Destination code. Messages are appended to the Current Messages string. The subVI, Keep N Lines, keeps several of the latest messages and throws away the older ones to conserve space in the indicator for current messages.

**Current Messages**, for direct display. This alarm handler would be called periodically by a high-level VI that contains the alarm message display.

There are other operations that an alarm handler might perform. If many other VIs need to display alarm messages or status information, you could have the alarm handler copy messages to a global variable for reading and display elsewhere. The handler could also call subVIs or set special global flags that cause some control operations to occur.

Another way to manage alarms is to use a global database as a repository for all your data. Each I/O point is represented by a cluster of information, and part of that information is a set of flags that represents alarm limits and alarm status flags. Alarm generators (probably part of the I/O handlers) set the alarm status flags based on the current value and the desired limits. Since the database is globally accessible, any VI can read the status of any alarm and take appropriate action.

All of this is limited by the real-time performance of the database, so you should approach with caution. If you use DSC, all aspects of alarm handling described here will be taken care of automatically.

### Techniques for operator notification

The fun part of this whole alarm business is notifying the operator. You can use all kinds of LabVIEW indicators, log messages to files, make sounds, or use external annunciator hardware. Human factors specialists report that a consistent and well-thought-out approach to alarm presentation is vital to the safe operation of modern control systems. Everything from the choice of color to the wording of messages to the physical location of the alarm readouts deserves your attention early in the design phase. When automatic controls fail to respond properly, it's up to the operator to take over, and he or she needs to be alerted in a reliable fashion.

Boolean indicators are simple and effective alarm annunciators. Besides the built-in versions, you can paste in graphics for the True and/or False cases for any of the boolean indicators. Attention-getting colors or shapes, icons, and descriptive text are all valuable ideas for alarm presentation.

You can log alarm messages to a file to provide a permanent record. Each time an alarm is generated, the alarm handler can call a subVI that adds the time and date to the message then appends that string to a preexisting text file. Useful information to include in the message includes the tag name, the present value, the nature of the alarm, and whether the alarm has just occurred or has been cleared. This same file can also be used to log other operational data and important system events, such as cycle start/stop times, mode changes, and so forth. DSC stores alarms and other events in an event log file. SubVIs are available to access and display information stored in the event log.

The same information that goes to a file can also be sent directly to a printer. This is a really good use for all those old dot-matrix serial printers you have lying around. Since the printer is just a serial instrument, it's a simple matter to use the **Serial Port Write** VI to send it an ASCII string. If you want to get fancy, look in your printer's manual and find out about the special escape codes that control the style of the output. You could write a driver VI that formats each line to emphasize certain parts of the message. As a bonus, dot-matrix printers also serve as an audible alarm annunciator if located near the operator's station. When the control room printer starts making lots of noise, you know you're in for some excitement.

Audible alarms can be helpful or an outright nuisance. Traditional control rooms and DCSs had a snotty-sounding buzzer for some alarms,

and maybe a big bell or Klaxon for real emergencies. If the system engineer programs too many alarms to trigger the buzzer, it quickly becomes a sore point with the operators. However, sound does have its place, especially in situations where the operator can't see the display. You could hook up a buzzer or something to a digital output device or make use of the more advanced sound recording and playback capabilities of your computer (see Chapter 20, "Data Visualization, Imaging, and Sound"). LabVIEW has a set of sound VIs that work on all platforms and serve as annunciators.

Commercial **alarm annunciator panels** are popular in industry because they are easy to understand and use and are modestly priced. You can configure these units with a variety of colored indicators that include highly visible labels. They are rugged and meant for use on the factory floor. Hathaway/Beta Corporation makes several models, ranging from a simple collection of lamps to digitally programmed units.

That covers sight and sound; what about our other senses? Use your imagination. LabVIEW has the ability to control most any actuator. Maybe you could spray some odoriferous compound into the air or dribble something interesting into the supervisor's coffee.

## Bibliography

- Boyer, S. A.: *SCADA: Supervisory Control and Data Acquisition*, ISA Press, Raleigh, N.C., 1993.
- Bryan, Luis A., and E. A. Bryan: *Programmable Controllers: Theory and Implementation*, C Industrial Text Company, Atlanta, GA, 1988.
- Corripio, Armando B.: *Tuning of Industrial Control System*, ISA Press, Raleigh, N.C., 1990.
- Hughes, Thomas A.: *Measurement and Control Basics*, ISA Press, Raleigh, N.C., 1988.
- Johnson, Gary (Ed.): *LabVIEW Power Programming*, McGraw-Hill, New York, 1998.
- McMillan, Gregory K.: *Advanced Temperature Control*, ISA Press, Raleigh, N.C., 1995.
- Montgomery, Douglas C.: *Introduction to Statistical Quality Control*, Wiley, New York, 1992.
- Shinskey, F. G.: *Process Control Systems*, McGraw-Hill, New York, 1988.
- Travis, Jeffrey: *Internet Applications in LabVIEW*, Prentice-Hall, Upper Saddle River, N.J., 2000.
- Wheeler, Donald J., and D. S. Chambers: *Understanding Statistical Process Control*, 2d ed., SPC Press, Knoxville, Tenn., 1992.

## Physics Applications

*Physics is Phun*, they told us in Physics 101, and, by golly, they were right! Once we got started at Lawrence Livermore National Laboratory (LLNL), where there are plenty of physics experiments going on, we found out how interesting the business of instrumenting such an experiment can be. One problem we discovered is just how little material is available in the way of instructional guides for the budding diagnostic engineer. Unfortunately, there isn't enough space to do a complete brain dump in this chapter. What we will pass along are a few references: Bologna and Vincelli (1983); and Mass and Brueckner (1965). Particularly, we've gotten a lot of good tips and application notes from the makers of specialized instruments (LeCroy Corporation 2000). Like National Instruments, they're all in the business of selling equipment, and the more they educate their customers, the more equipment they are likely to sell. So, start by collecting catalogs and look for goodies like sample applications inside. Then, get to know your local sales representatives, and ask them how to use their products. Having an experienced experimental physicist or engineer on your project is a big help, too.

We're going to treat the subject of physics in its broadest sense for the purpose of discussing LabVIEW programming techniques. The common threads among these unusual applications are that they use unconventional sensors, signal conditioning, and data acquisition equipment, and often involve very large data sets. Even if you're not involved in physics research, you are sure to find some interesting ideas in this chapter.

Remember that the whole reason for investing in automated data acquisition and control is to improve the quality of the experiment. You can do this by improving the quality and accuracy of the recorded data

and by improving the operating conditions of the experiment. Calibrating transducers, instruments, and recording devices as well as monitoring and stabilizing the physical parameters of the experiment all lead to better results. Having a flexible tool such as LabVIEW makes these goals much easier to achieve than in the past. Twenty years ago, we had computers, but they were so cumbersome to configure that the researcher needed a large staff just to support simple data acquisition. That meant less money and time available for improving the experiment, examining the data, and doing physics.

## Special Hardware

In stark contrast to ordinary industrial situations, physics experiments are, by their nature, involved with exotic measurement techniques and apparatus. We feel lucky to come across a plain, old low-frequency pressure transducer or thermocouple. More often, we're asked to measure microamps of current at high frequencies, riding on a 35-kV dc potential. Needless to say, some fairly exotic signal conditioning is required. Also, some specialized data acquisition hardware, much of which is rarely seen outside of the physics lab, must become part of the researcher's repertoire. Thankfully, LabVIEW is flexible enough to accommodate these unusual instrumentation needs.

### Signal conditioning

High-voltage, high-current, and high-frequency measurements require specialized signal conditioning and acquisition equipment. The sources of the signals, though wide and varied, are important only as far as their electrical characteristics are concerned. Interfacing the instrument to the data acquisition equipment is a critical design step. Special amplifiers, attenuators, delay generators, matching networks, and overload protection are important parts of the physics diagnostician's arsenal.

**High-voltage measurements.** To measure high-voltage signals, you will need to reduce the magnitude of the signal to something that your analog to digital converter (ADC) can safely and accurately accommodate. For most signals, resistive voltage dividers are the easiest to use. You can buy commercial high-voltage probes of several types. Those intended for routine ac and dc voltmeters, such as those made by Fluke-Phillips, are sufficient for frequencies up to about 60 Hz. Special high-voltage oscilloscope probes such as the Tektronix P6015 are available with divider ratios of 10, 100, and 1000 to 1 and are useful up to 20 kV dc, at frequencies up to 75 MHz, and with input impedances of 10 to 100 M $\Omega$ . All you have to do is match the output impedance (and capacitance, for

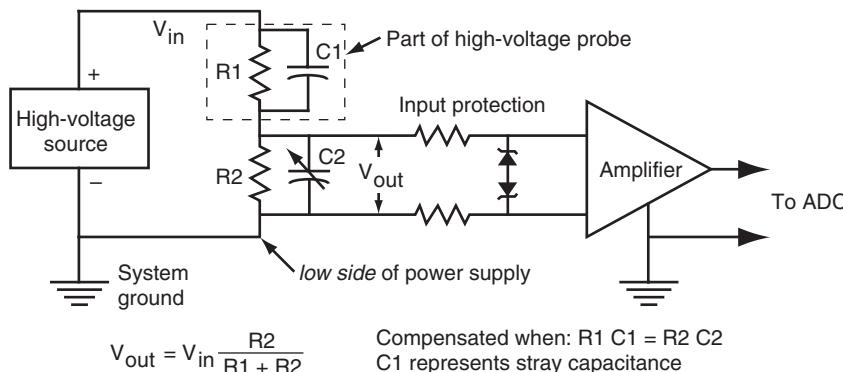
high frequencies) to that of your data acquisition equipment, perhaps by using a suitable amplifier. A 'scope probe with a  $1\text{-M}\Omega$  resistor for termination works fine with a plug-in data acquisition board.

For measurements up to hundreds of kilovolts, you can buy high-voltage probes from Ross Engineering and others. These probes are compact (considering their voltage ratings) and are intended for permanent installation. As with 'scope probes, you have to match the output impedance for highest accuracy (Figure 19.1).

Pulsed high voltages can be challenging because of their extra high-frequency content. Resistive probes incorporate frequency compensation capacitors to flatten frequency response (as always, proper matching at the output is paramount). Capacitive dividers and custom-made pulse transformers may be optimum in certain cases.

If the measurement you are making is part of a floating system (that is, it is not referenced to ground), you will also need an **isolation amplifier**. Most commercial isolation amplifiers operate only at lower voltages, so they must be inserted after a voltage divider. Sometimes, you can float an entire measurement system at high voltage. This requires an isolated power supply for the electronics and typically incorporates a fiber-optic link for communications. Many accelerator experiments have been constructed around such systems.

**Safety** is your number one concern in high-voltage system design. First, protect the personnel, then protect the equipment, then protect the data. Floating systems are probably the most dangerous because they tempt you to reach in and turn a knob—a sure way to an early grave. Cover all high-voltage conductors with adequate insulation, and mark all enclosures with warning labels. Enclosures should be interlocked with



**Figure 19.1** General application of a high-voltage probe. Frequency compensation is only needed for ac measurements. Always make sure that your amplifier doesn't load down the voltage divider.

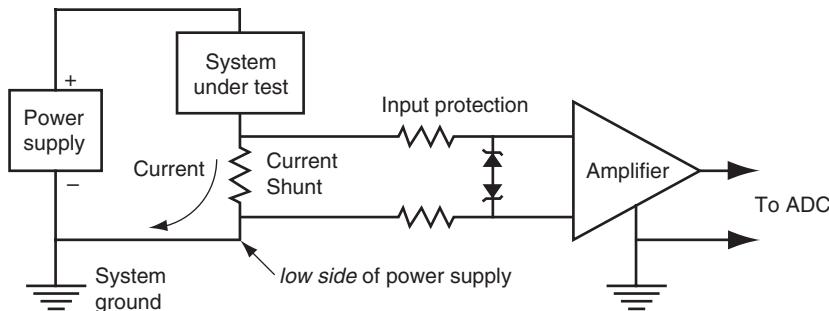
safety devices that shut down and *crowbar* (short-circuit) the source of high voltage. Cables emerging from high-voltage equipment must be properly grounded to prevent accidental energization. At the input to your signal conditioning equipment, add overvoltage protection devices such as current-limiting resistors, zener diodes, Transorbs, metal-oxide varistors (MOVs), and so forth. Protect *all* inputs, because improper connections and transients have a way of creeping in and zapping your expensive equipment . . . and it always happens two minutes before a scheduled experiment.

**Current measurements.** It seems that we spend more time measuring electrical currents than almost anything else in the physics lab. Whether it's high or low current, ac or dc, the measurement is always more complex than expected.

AC and pulse currents are often the easiest to measure because you can use a **current transformer**. Simple robust devices, current transformers rely on a dynamic magnetic field to couple a fraction of the measured current in the primary winding into the secondary winding. Pearson Electronics makes a wide variety of solenoidal current transformers, usable up to tens of thousands of amps and 20 MHz. The output has an impedance of  $50\ \Omega$ , so it matches well to wideband amplifiers and digitizer inputs. There are two limitations to current transformers. First, DC current tends to prematurely saturate the core, resulting in distortion and other amplitude errors. Second, these transformers are naturally ac-coupled, so you have to be aware of the low-frequency response limit of your transformer. Specifically, dc and low-frequency ac information will be lost. On the other hand, there is no concern about contact with high dc voltages unless the transformer insulation breaks down. Note that the output of a current transformer is an image of the current waveform, not an equivalent rms or average value. Therefore, you must record *waveforms*, not low-speed dc signals.

**Hall Effect devices** are used to make clamp-on dc-coupled current probes. These solid-state devices respond to the instantaneous magnetic field intensity surrounding a conductor, producing a proportional output. One commercial Hall Effect current probe is the Tektronix AM503S. With it, you can measure AC or DC currents from millamps to 500 A at frequencies up to 50 MHz, and to do this you merely clamp the probe around the conductor. Though expensive, they are real problem solvers.

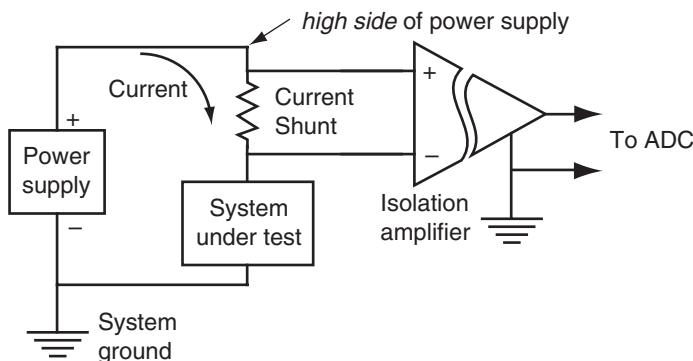
**Current shunts**, also known as **current viewing resistors**, are the simplest, and often cheapest, current transducers, consisting of a resistor that is placed in series with the source of current. The resulting voltage drop follows Ohm's Law, where the voltage is equal to the current multiplied by the resistance. You can make your own shunt for low currents by inserting a low-value resistor in the circuit. For high currents,



**Figure 19.2** Signal conditioning for a current shunt installed on the preferred side of the system under test.

up to thousands of amps, there are commercially made shunts available. Standard outputs are usually 50 or 100 mV at full current, so you may need an amplifier if your ADC system can't handle low-level signals. Try to connect the shunt on the *low side*, or ground return, of the power source to eliminate the common-mode voltage (Figure 19.2). Note that this method is only feasible if the system under test or the power supply can be isolated from ground. If you must wire the shunt in series with the *high side* of the power source, you need an isolation amplifier (Figure 19.3). Unfortunately, isolation amplifiers are expensive, and they generally don't have very wide bandwidths.

**High-frequency measurements.** Many experiments involve the detection of pulsed phenomena, such as nuclear particle interactions, lasers,



**Figure 19.3** This example shows an isolation amplifier with a high side current shunt. The common-mode voltage is that which appears across the system under test, and may exceed the capability of nonisolated amplifiers.

and explosions. The detector may sense ion currents or some electrical event in a direct manner, or it may use a multistep process, such as the conversion of particle energy to light, then light to an electrical signal through the use of a photomultiplier or photodiode. Making quantitative measurements on fast, dynamic phenomena is quite challenging, even with the latest equipment, because there are so many second-order effects that you must consider. High-frequency losses due to stray capacitance, cable dispersion, and reflections from improperly matched transmission lines can severely distort a critical waveform. It's a rather complex subject. If you're not well versed in the area of pulsed diagnostics, your best bet is to find someone who is.

There are a number of instruments and ancillary devices that you will often see in high-frequency systems. For data acquisition, you have a choice of acquiring the entire waveform or just measuring some particular characteristic in real time. Waveform acquisition implies the use of a fast **transient digitizer** or digitizing oscilloscope, for which you can no doubt find a LabVIEW driver to upload the data. Later in this chapter, we'll look at waveform acquisition in detail. Some experiments, such as those involving particle drift chambers, depend only on time interval or pulse coincidence measurements. Then you can use a time interval meter, or **time-to-digital converter (TDC)**, a device that directly measures the time between events. Another specialized instrument is the **boxcar averager, or gated integrator**. This is an analog instrument that averages the signal over a short gate interval and then optionally averages the measurements from many gates. For periodic signals, you can use a boxcar averager with a low-speed ADC to reconstruct very high frequency waveforms.

## CAMAC

**CAMAC** (Computer Automated Measurement and Control) is an old, reliable, and somewhat outdated standard for data acquisition in the world of high-energy physics research. It remains a player in research labs all over the world. LabVIEW has instrument drivers available for use with a variety of CAMAC instruments and controllers, particularly those from Kinetic Systems and LeCroy. Because other hardware platforms have replaced CAMAC, we're no longer going to say much about it. If you want to read more, you can probably find a previous edition of this book in which there's a long CAMAC section.

## Other I/O hardware

There are some other I/O and signal-conditioning standards that you may run into in the world of physics research. And, of course, there really is no limit on what you can use; it's just that there are some specialized

functions that have been implemented over the years that aren't always commonly available.

**FASTBUS.** FASTBUS (ANSI/IEEE 960-1986) represents the fastest, highest-density data acquisition hardware available today. It is a modular standard featuring a 32-bit address and data bus, an asynchronous ECL backplane capable of 80 MHz, and the ability to use multiple masters or controllers. It's in fairly wide use in high-energy physics at such institutions as CERN in Switzerland. It's a sophisticated I/O system, but it's not for the casual user or those with miniature budgets. But if you get involved with the right laboratory, you may well see FASTBUS equipment. Currently, you can interface it to a LabVIEW system by using CAMAC as an intermediary or with a PCI bus interface, either of which is perfectly acceptable.

**VME.** The VME bus is a high-performance, general-purpose computing platform. It's an open standard with modules available from hundreds of manufacturers offering state-of-the-art CPUs and peripherals as well as a wide variety of I/O hardware. Interfaces for everything from analog I/O to image processing hardware to exotic military test equipment is available in VME format. You can use LabVIEW on VME-based CPUs that run a suitable operating system, such as Windows. Or, you can use an MXI interface to connect a desktop machine to a VME backplane. An example is the National Instruments VMEPCI8000, which connects a PCI-bus computer to a B-size VME system, making your computer appear as if it were plugged into the VME backplane. Your computer can directly access the VME address space, and VME bus masters can directly access the computer's memory and resources. Up to eight MXI-based devices, such as VXI crates, can be daisy-chained into such a system. To access devices on the VME bus, you use NI-VXI/VISA drivers. VISA and VXI drivers are discussed in detail in Chapter 11, "Instrument Driver Development Techniques."

**VXI.** VXI is taking over the title of workhorse data acquisition interface for physics applications, particularly when high-performance transient digitizers are required. It's well planned and supported by dozens of major manufacturers, and offers many of the basic functions you need. And, of course, there is excellent LabVIEW support. Drivers are available for many instruments and PCs can be installed right in the crate.

**NIM.** Another older standard still in use in the physics community is the **NIM (Nuclear Instrumentation Manufacturers)** module format, originally established in 1964. It's not an I/O subsystem like CAMAC or VXI; instead, it's a modular signal conditioning standard.

Modules are either full- or half-height and plug into a powered NIM bin with limited backplane interconnections. Because the CAMAC format was derived from the earlier NIM standard, NIM modules can plug into a CAMAC crate with the use of a simple adapter.

Many modern instruments are still based on NIM modules, particularly nuclear particle detectors, pulse height analyzers, and boxcar averagers such as the Stanford Research SR250. We still use them in the lab because there are so many nice functions available in this compact format, such as amplifiers, trigger discriminators, trigger fan-outs, and clock generators.

## Field and Plasma Diagnostics

Among the many low-speed applications of LabVIEW in physics are those that deal with **field mapping** and **DC plasma diagnostics**. These applications combine data acquisition with motion control for field mapping and the generation of ramped potentials for plasma diagnostics. There are many variations on these experiments, so we'll just cover a few simple applications that might give you some insight as to how LabVIEW might help out in your lab.

### Step-and-measure experiments

When you have a system that generates a static field or a steady-state beam, you probably will want to map its intensity in one, two, or three dimensions and maybe over time as well. We call these **step-and-measure** experiments because they generally involve a cyclic procedure that moves a sensor to a known position, makes a measurement, moves, measures, and so forth, until the region of interest is entirely mapped. Some type of **motion control** hardware is required, along with a suitable probe or sensor to detect the phenomenon of interest.

**Motion control systems.** There are many actuators that you can use to move things around under computer control—the actuators that make robotics and numerically controlled machines possible.

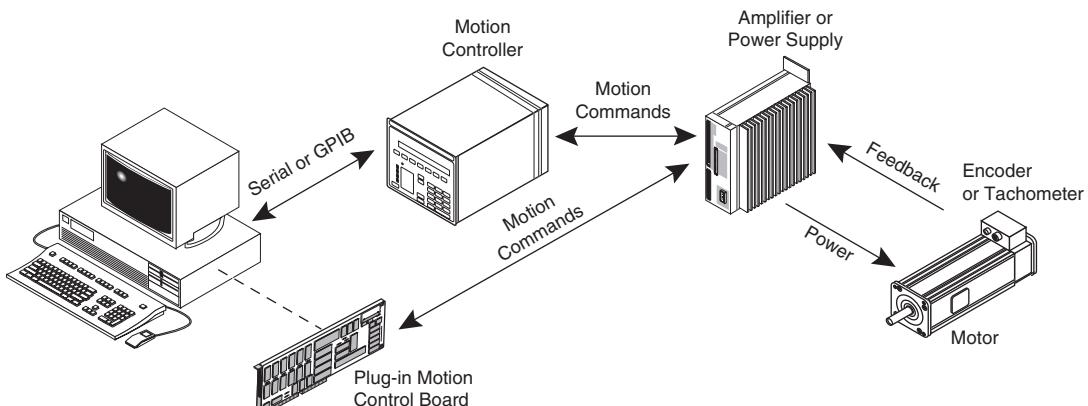
For simple two-position operations, a pneumatic cylinder can be controlled by a solenoid valve that you turn on and off from a digital output port. Its piston moves at a velocity determined by the driving air pressure and the load. Electromagnetic solenoids can also move small objects back and forth through a limited range.

But more important for your step-and-measure experiments is the ability to move from one location to another with high precision. Two kinds of motors are commonly used in these applications, **stepper motors** and **servo motors**. Think of a stepper motor as a kind of

digital motor. It uses an array of permanent magnets and coils arranged in such a way that the armature (the part that moves) tends to snap from one location to the other in response to changes in the polarity of the DC current that is applied to each coil. Stepper motors are available with anywhere from 4 to 200 steps per revolution. Linear stepper motors are also available. Steppers have the advantage of simplicity and reasonable precision. Their main disadvantages are that you can't position them with infinite resolution and their speed is somewhat limited, typically 1000 to 2000 steps per second. There is one option for increasing resolution, called **microstepping**. With microstepping, the effective steps per revolution can be increased 10 times, and sometimes more. This requires special control hardware and sometimes compromises the performance of the motor with regard to torque.

Servomotors are similar to ordinary motors except that they are intended for variable-speed operation, down to and including stalled, for indefinite periods. Both AC and DC servo motors are in wide use. A servo system also includes a positional or velocity feedback device, such as a shaft encoder or tachometer, to precisely control the motion of the shaft. (Such feedback may also be used with stepper motors.) With the use of feedback, servo motor systems offer extremely high precision and the ability to run at high speed. Their disadvantage is the requirement for somewhat complex control circuitry.

The next item you need to select is a motion controller (Figure 19.4). It may take the form of a packaged system with RS-232 GPIB; or Ethernet communications from companies such as Gallil, Parker Compumotor, or Klinger; or as plug-in boards for the PC from companies such as National Instruments and Parker Compumotor. Motor controllers are

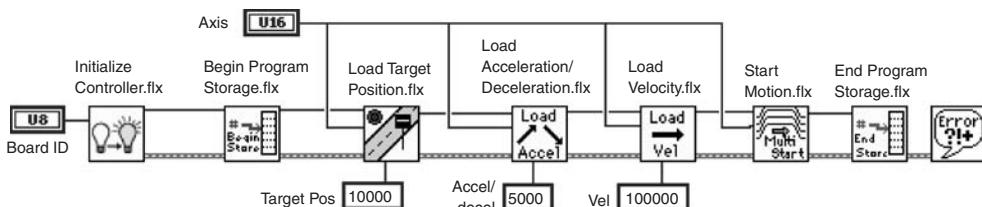


**Figure 19.4** A typical motion control system consists of a computer, a controller (internal or external), an amplifier or power supply, and a motor, perhaps with an encoder or tachometer for feedback.

also available in various modular formats such as CAMAC, VME, and VXI. LabVIEW drivers are available for many of these devices; we'll discuss some of them later. With any motor, you also need a power supply—also known as an *amplifier*—which is a power electronics package to drive the coils of the motor in an appropriate fashion. The amplifier supplies current to the motor, provides signal conditioning for encoders or tachometers, and isolates the sensitive I/O ports of the motion controller from the harsh external world.

About the easiest way to integrate motion control into a LabVIEW application is to use the plug-in motion control boards and drivers from National Instruments. Several years ago, National Instruments acquired **nuLogic**, a company famous for its Macintosh- and Windows-compatible motion boards and LabVIEW drivers. Boards are available that support up to four axes of steppers and servos, in PCI, Compact-PCI, and PXI formats and a stand-alone controller with a FireWire (IEEE-1394) interface. The company also offers a line of stepper and servo power supplies, or you can wire the boards up to another manufacturer's power supply. Most of the boards also include general-purpose analog and digital I/O.

The high-performance motion boards (FlexMotion 7344 series) are highly programmable, with onboard real-time processors that allow you to write sophisticated motion routines and have them execute without intervention from the host computer. Figure 19.5 shows an example of a FlexMotion application. As with any programmable motion controller, you can load a sequence of commands into a buffer on the FlexMotion board and then have it execute that sequence autonomously. One of the first VIs in the sequence, Begin Program Storage, tells the board that all subsequent commands are to be stored, not executed immediately. Several trajectory commands are sent next, including acceleration, deceleration, velocity, and so forth. Finally the End Program Storage VI tells the board to terminate command storage. The sequence can now be initiated through software by calling the Run Program VI, or you can configure the board to run the program when a hardware trigger occurs.



**Figure 19.5** This example stores a motion sequence for later execution on a FlexMotion board. The stored program can be quite complex, and it can run without intervention from LabVIEW.

A few other motion control systems are also supported by LabVIEW drivers. In the driver library, you'll find VIs for several models from **Aerotech Unidex** and **Newport Corporation**. For other products, check with your favorite manufacturer.

**Reading encoders.** A recurring theme in the world of motion control is reading an **encoder**. Encoders can be of the **incremental** or **absolute** type. Incremental encoders produce a continuous train of pulses when moved while absolute encoders have a parallel output that directly tells you the absolute position of a shaft or linear actuator. Incremental encoders typically have two outputs that are 90 degrees out of phase, in which case they are called *quadrature* encoders. The phase shift permits determination of direction, assuming that you have the proper readout logic.

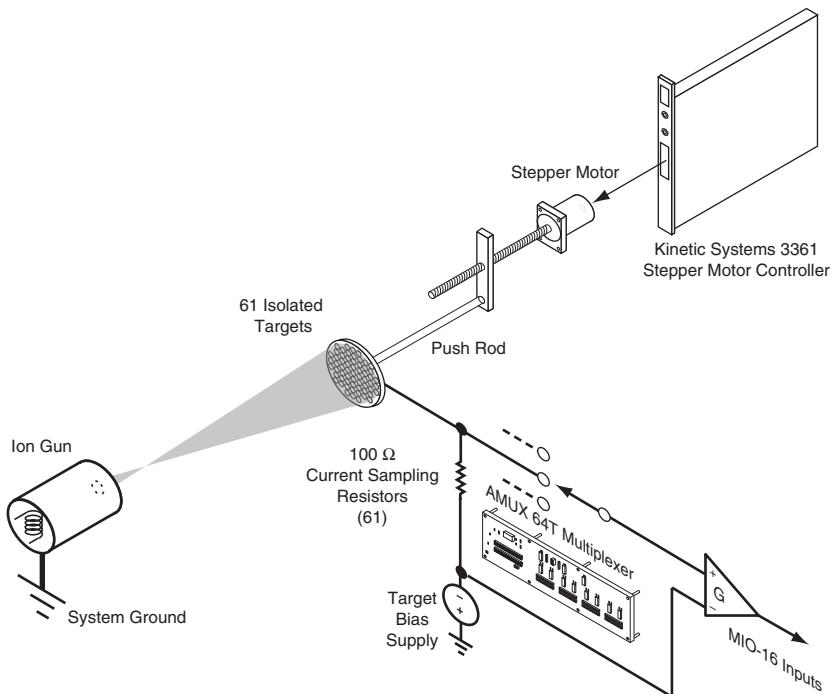
To read a quadrature encoder with LabVIEW, you have a few choices. First, you can use any of the motion control products described previously. The driver VIs always have a way to read the encoder position, and they also typically include direct measurement of velocity. Second, you can try to use a special-purpose integrated circuit that converts quadrature pulses to a format suited to a regular DAQ board with counter-timers. Application note AN084, **Using Quadrature Encoders with E Series DAQ Boards**, shows how it's done. If that sounds troublesome, consider buying the BNC-2120 accessory for an E-series DAQ board. It includes the necessary circuitry, and there's an example VI that does the measurement.

Here's a better solution: Use one of the timing I/O boards from National Instruments. The 660x family of counter-timer boards has an exceptionally versatile chip onboard, the NI-TIO, that has the ability to directly interpret quadrature encoder signals, among other things. On National Instruments' Web site you'll find the handy **Position Meas Quad Encoder VI**. All you have to do is connect the two quadrature signals from your encoder to the *source* and *up/down* pins of one of the counters on your TIO board. Many encoders also have a **Z index pulse** output that is asserted only once per revolution; it's useful for obtaining an absolute orientation on a rotational measurement. If you have a Z index, wire it to the *gate* input of the counter. That's all there is for signal connections.

The VI takes advantage of some advanced features of the TIO boards. First, you can activate low-pass filtering on the signal input lines to reduce noise susceptibility. Second, you can activate **resolution multiplication**, where the hardware interpolates the encoder pulses by a factor of 2 or 4. The output of the VI is the measured position in counts, which you will probably want to scale to degrees or some other useful unit.

**Motion application: An ion beam intensity mapper.** One lab that Gary worked in had a commercially made ion beam gun that we used to test the sputter rate of various materials. We had evidence that the beam had a nonuniform intensity cross section (the  $xy$  plane), and that the beam diverged (along the  $z$  axis) in some unpredictable fashion. In order to obtain quality data from our specimens, we needed to characterize the beam intensity in  $x$ ,  $y$ , and  $z$ . One way to do this is to place sample coupons (thin sheets of metal or glass) at various locations in the beam and weigh them before and after a timed exposure to obtain relative beam intensity values. However, this is a tedious and time-consuming process that yields only low-resolution spatial data. Preliminary tests indicated that the beam was steady and repeatable, so high-speed motion and data acquisition was not a requirement.

Our solution was to use a plate covered with 61 electrically isolated metal targets (Figure 19.6). The plate moves along the  $z$  axis, powered by a stepper motor that drives a fine-pitch lead screw. The stepper motor is powered by a Kinetic Systems 3361 Stepper Motor Controller CAMAC module. Since the targets are arranged in an  $xy$  grid, we can

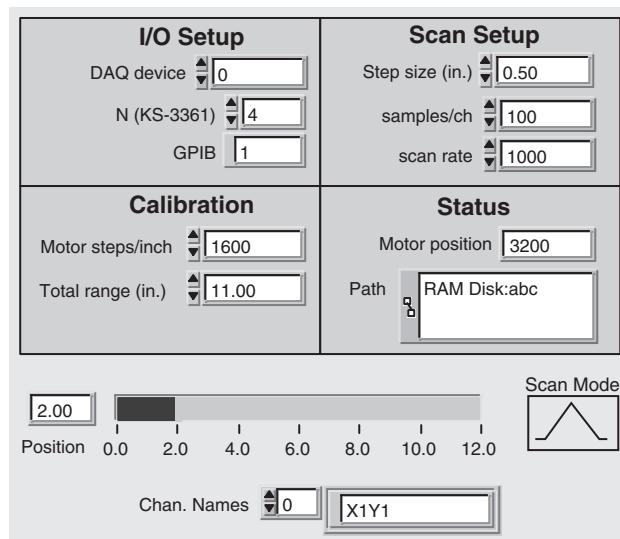


**Figure 19.6** The ion beam intensity mapper hardware. The plate holding 61 isolated probes moves along the axis of the ion beam. Current at each probe is measured at various positions to reconstruct a 3D picture of the beam intensity.

obtain the desired  $xyz$  mapping by moving the plate and taking data at each location. Each target is connected to a negative bias supply through a  $100\text{-}\Omega$  current-sensing resistor, so the voltage across each resistor is  $100 \text{ mV/mA}$  of beam current. Current flows because the targets are at a potential that is negative with respect to the (positive) ion gun, which is grounded. A National Instruments AMUX-64T samples and multiplexes the resulting voltages into an NB-MIO-16 multifunction board. The bias voltage was limited to less than  $10 \text{ V}$  because that's the common-mode voltage limit of the board. We would have liked to increase that to perhaps  $30\text{--}50 \text{ V}$  to collect more ions, but that would mean buying 61 isolation amplifiers.

The front panel for this experiment is shown in Figure 19.7. Two modes of scanning are supported: unidirectional or bidirectional (out and back), selectable by a boolean control (**Scan Mode**) with pictures pasted in that represent these motions. A horizontal fill indicator shows the probe position as the scan progresses to cover the desired limits.

Examine the diagram of the main VI in Figure 19.8. An overall While Loop keeps track of the probe location in a shift register, which starts at zero, increases to a limit set by **Total Range**, then steps back to zero if a bidirectional scan is selected. The VI stops when the scan is finished then returns the probe to its starting location. Arithmetic in the upper half of the loop generates this stepped ramp.



**Figure 19.7** Front panel of the ion beam intensity scan VI. The user runs the VI after setting all the parameters of the scan. A horizontal fill indicator shows the probe position.

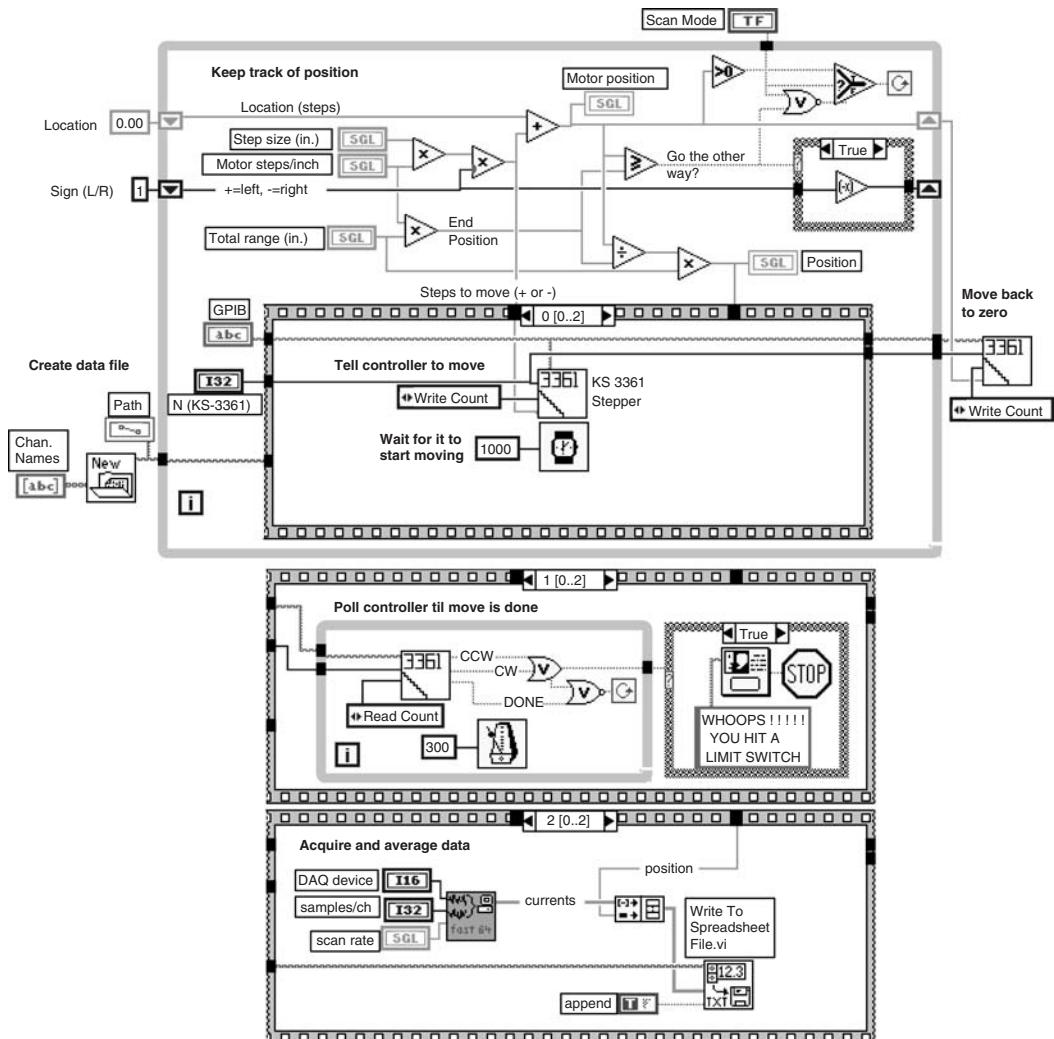


Figure 19.8 Diagram of the ion beam intensity scan VI. The While Loop executes once for each step of the positional ramp. The sequence moves the probe and acquires data.

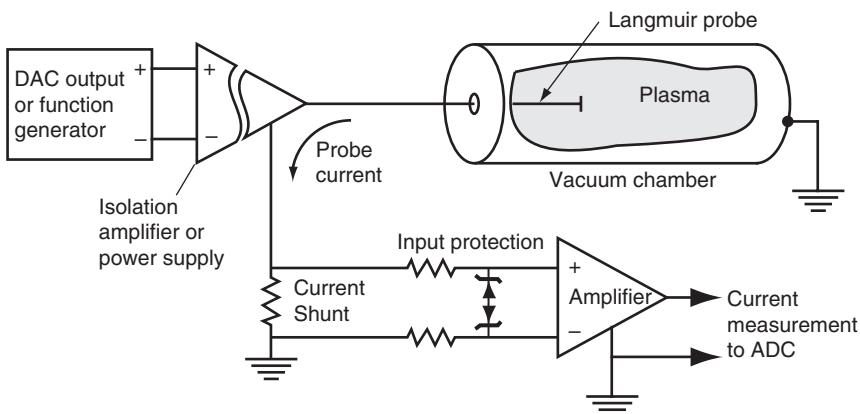
The Sequence structure inside the main loop has three frames. Frame zero commands the motor controller to move to the new position. Frame one polls the motor controller, awaiting the *done* flag, which indicates that the move is complete. If the VI encounters a limit switch, it tells the user with a dialog box and execution is aborted. Frame two acquires data by calling a subVI that scans the 64 input channels at high speed and averages the number of scans determined by a front panel control. Finally, the position and the data are appended to a data file in tab-delimited text format for later analysis. As always, the

format of the data was determined beforehand to assure compatibility with the analysis software.

### Plasma potential experiments

The plasma we're talking about here doesn't flow in your veins. This plasma is the so-called fourth state of matter, where most of all the atoms are ionized. Plasma is formed by depositing sufficient energy into a substance (usually a gas) to rip away one or more electrons from the atoms, leaving them positively charged. The ionizing energy may come from an electrical discharge (an electron bombardment) or some form of ionizing radiation, such as light, gamma rays, or nuclear particles. The ions and electrons in a plasma love to recombine and react with other materials, so most plasma experiments are performed in a vacuum chamber with a controlled atmosphere containing only the desired gaseous species.

Some parameters that we like to determine in experiments are the plasma space potential, electron temperature, floating potential, and ion and electron densities. A very simple plasma diagnostic technique, the **Langmuir probe**, makes most of these measurements simple. The method involves the measurement of ion and/or electron current flowing in a small metal probe that is positioned in contact with the plasma. By varying the voltage (potential) applied to the probe and measuring the current, a curve called the *probe characteristic* is acquired. All you need is a variable voltage source, a sensitive current monitor, and a recording mechanism (anyone for LabVIEW?). Figure 19.9 shows how a simple Langmuir probe experiment might be connected.



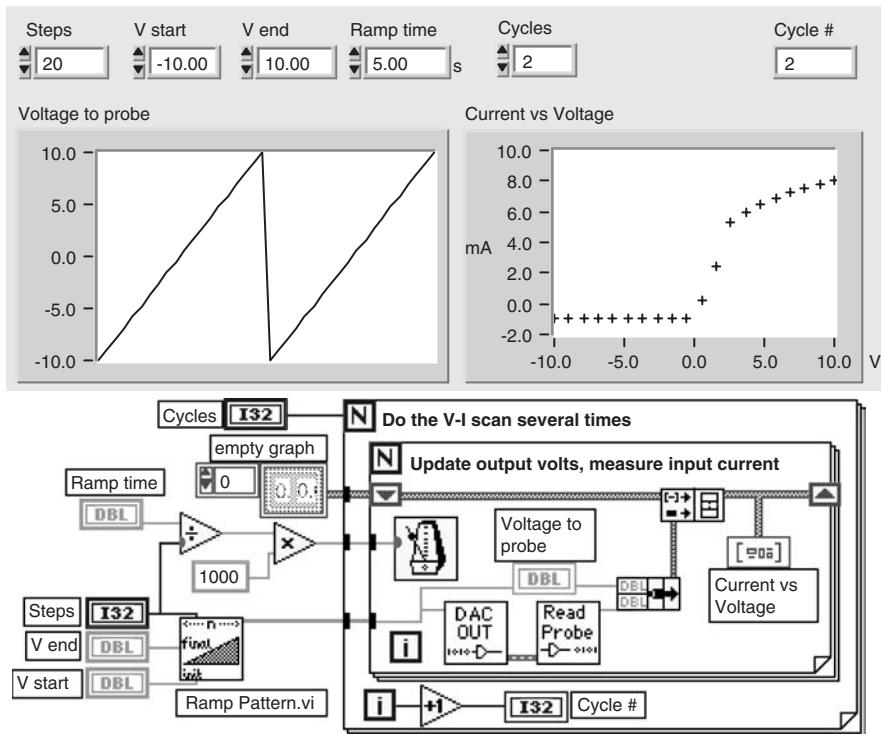
**Figure 19.9** Basic electrical connections for a Langmuir probe experiment. The probe voltage is supplied by an isolation amplifier, programmable power supply, or function generator with a large output voltage swing.

Electrically, the most important piece of equipment is the voltage source that drives the probe. Depending on the experiment, you may need a source that can produce anywhere from  $\pm 10$  V at 10 mA up to  $\pm 50$  V at 100 mA, or greater. Ordinary DACs, such as those built into a typical plug-in board, can only reach  $\pm 10$  V and a few mA, which may be sufficient for some laboratory experiments. For low-speed DC experiments, we like to use a laboratory power supply with an external analog input driven by a DAC. The output of such a supply is isolated from ground and is nearly bulletproof. For experiments in which you need a high-speed ramp waveform or pulse, you may need a function generator and a high-performance amplifier. As for the current-sensing function, pick a shunt resistor that only drops a fraction of a volt at the expected probe current ( $R = V/I$ ), and follow it with the usual differential amplifier on your DAQ board or signal conditioner. Input protection is a good idea because the plasma source generally has a high-voltage source. Accidental contact with any high-voltage elements (or arcing) will likely destroy the amplifier.

To obtain the Langmuir probe characteristic, your LabVIEW program will have to generate a ramp waveform. Steady-state plasma experiments can be performed at low speeds over a period of seconds or minutes, so it's practical for your program to calculate the desired voltage and drive an output device. Figure 19.10 shows a simple V-I scan experiment that uses a precalculated array of values representing a ramp waveform that is sent to a DAC repeatedly. You can use any conceivable waveform to initialize the data array. Nested For Loops step through the values. After each step, a measurement is taken from the system (for instance, the current from a Langmuir probe). Inside the inner For Loop the new voltage is written to the output, then the resulting probe current is measured. Error I/O links the DAC Out and Read Probe subVIs so that they execute in the proper order. If extra settling time is required before the measurement, add a suitable delay between the DAC Out and Read Probe VIs.

The XY Graph (**current versus voltage**) acts like a real-time *xy* recorder. Since LabVIEW doesn't have a built-in *xy* strip chart, you can simulate one by updating the graph each time the inner For Loop executes. A shift register carries the accumulated array of clusters, where the cluster contains *x* (voltage) and *y* (current) data. An empty graph constant initializes the shift register. If you don't care to see the graph update for every sample point, put it outside the inner For Loop and let the cluster array build itself on the border of the loop. This is much faster and more efficient, too.

The Read Probe subVI acquires and scales the current measurement. If you want to store the data on disk, that subVI would be a logical place



**Figure 19.10** A simple V-I scan experiment. This is another way of generating low-speed, recurrent ramps, using a precalculated array of output voltages. Values are sent to a digital to analog converter (DAC) repeatedly, and the response of the system under test is measured and recorded after each output update. The current versus voltage graph acts like a real-time xy chart recorder.

to write the data to a file. Outside the For Loops, you would open a suitable file and probably write some kind of header information. Remember to close the file when the VI is through.

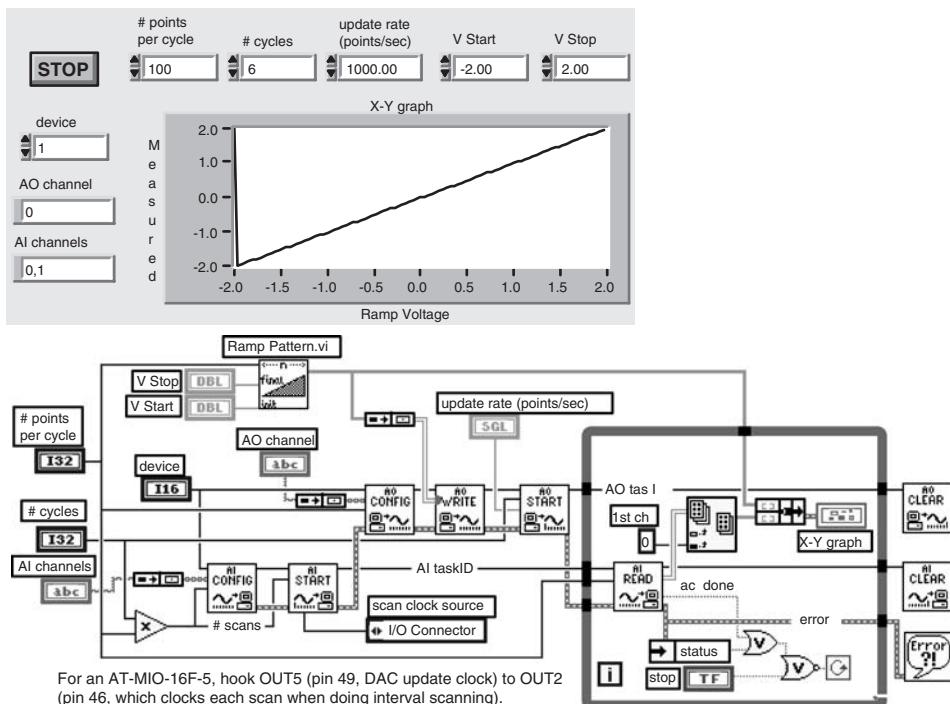
In faster experiments, LabVIEW may not have enough time to directly control the stimulus and response sequence. In that case, you can use a couple of different approaches, based on plug-in data acquisition boards or external, programmable equipment.

The data acquisition library supports buffered waveform generation as well as buffered waveform acquisition on plug-in boards with direct memory access (DMA). Essentially, you create an array of data representing the waveform, then tell the DMA controller to write the array to the DAC, one sample at a time, at a predetermined rate. Meanwhile, you can run a data acquisition operation also using DMA that is synchronized with the waveform generation. A set of examples is available in

the directory labview\examples\DAQ\analog-io. Synchronization of the input and output operations can be determined three possible ways.

- The DAC update clock is externally wired to an input (SCANCLK) which triggers a scan of the ADC. Each time a new step value in the ramp is generated, the ADC collects one value from each defined input channel (i.e., one scan).
- The ADC scans continuously, and its clock output triggers the DAC updates.
- An external input is supplied to both the ADC and DAC clocks. The signal can be derived from an onboard counter-timer set up as a pulse generator or from an external pulse generator.

The VI in Figure 19.11 uses the first method where DAC updates trigger ADC scans. The controls allow you to specify the ramp waveform's start and stop voltages, number of points, and the DAC update rate.



**Figure 19.11** Using the Data Acquisition library and an AT-MIO-16F-5 board, this VI simultaneously generates a ramp waveform and acquires data from one or more analog inputs. Only one external jumper connection is required; note that the pin number and signal name vary slightly between models of DAQ boards.

Multiple cycles can be generated, and each cycle is plotted independently. For the analog inputs, you can acquire data from one or more channels with adjustable scan rate and adjustable number of points to acquire. Only one channel is plotted in this example, but data from all scanned channels is available on the diagram.

This example makes effective use of the intermediate-level data acquisition functions. Two tasks are defined, one for input and one for output. Error I/O controls the execution sequence and, of course, handles errors. Synchronization of the input and output operations is determined by the DAC update clock which is externally wired to an input which clocks each scan of the ADC. Each time a new step value in the ramp is generated, the ADC collects one value from each defined input channel (i.e., one scan). Referring to the diagram, here are the details of the program's steps:

1. The **Ramp Pattern** VI (from the Signal Generation VIs in the Analysis library) creates an array of values corresponding to a ramp waveform. You could use other functions if you needed a different waveform.
2. **AI Config** sets up the analog input hardware on the board for the desired number of channels and allocates a data acquisition buffer that's sized for the number of scans to be acquired.
3. **AI Start** begins the buffered acquisition process that was defined by AI Config. The Trigger Type is set to 2 (I/O connector) which tells the board to use an external scan clock.
4. **AO Config** sets up the analog output hardware on the board and allocates a data buffer sized for the number of points to generate.
5. **AO Write** loads the ramp waveform into the output buffer in preparation for waveform generation.
6. **AO Start** starts the buffer waveform generation process that was defined by AO Config. The number of buffer iterations is set by the **Number of Cycles** control, meaning that the waveform will be produced several times.
7. **AI Read** fetches the acquired data from the buffer and makes it available to the XY Graph, where the measured voltage is plotted versus the calculated scan ramp. AI Read makes an estimate (based on the number of points to acquire and the sample rate) of how long to wait for data before timing out, so your program won't get stuck if there is a hardware failure.
8. The While Loop will continue executing the AI Read until an error occurs or the user clicks the **Stop** button.

9. Always clean up your mess. **AI Clear** and **AO Clear** terminate any I/O in progress and release their respective memory buffers. No error checking is done here because these functions can do no harm, even if no AI or AO operations are in progress. The **Simple Error Handler** checks for and reports any data acquisition errors.

In this example, where the analog output is updated first, there is a very short time delay before the ADC acquires a sample. Your experiment may not have time to settle, resulting in an unpredictable error. An alternative is to acquire a new reading from the ADC and then update the DAC output. This maximizes settling time and is probably a better overall solution.

For the E-series boards, a multitude of internal soft connections are possible, thanks to some elaborate onboard switching. This makes it easier to configure an E-series board for flexible scanning, like our synchronization problem. The **Synchronize AI AO for E-Series Boards** VI obtains the ADC scan clock from the DAC update clock in a manner similar to the example in Figure 19.11. Multiple AI and AO channels are supported.

Another way to solve this problem is to use a function generator for the stimulus waveform and a plug-in board or external digitizing instrument for the measurement. A great way to synchronize the stimulus and measurement operations is to use a time-base clock. If you can get your hands on a modern **arbitrary waveform generator**, you will find that it has a clock output. Each time the instrument updates its DAC output, a pulse is delivered to the clock output. This clock can then be applied to the external clock input of your data acquisition board or digitizer. If the signal generator is programmed to produce 1024 clocks in one cycle of the waveform, your digitizer is guaranteed to acquire exactly 1024 ADC samples. Nice and neat, and always in phase. To see how to program your DAQ board, look at the DAQ example VIs **Acquire N Scans-ExtChanClk** and **Acquire N Scans-ExtScanClk**.

If you can't use a clock, then there may be some way to use a trigger signal. Even simple analog function generators have a trigger output that occurs when the waveform starts. Use that to start a data acquisition operation or trigger a digital oscilloscope (see the section on **triggering** which follows). All you have to do is make sure that the period of the waveform is a little longer than the DAQ operation. Also, as with any external signal generation, you may want to dedicate a second ADC channel to monitoring the stimulus waveform, just to be sure that the amplitude is what you think it is.

## Handling Fast Pulses

Pulse and transient phenomena abound in the world of physics research. As we mentioned before, there is a host of specialized instrumentation to accompany the various detectors that produce transient waveforms. In this section, we'll look at some of those instruments, the LabVIEW programs that support them, and some of the programming tricks that may be of general use.

### Transient digitizers

A whole class of instruments, called **transient digitizers**, have been developed over the years to accommodate pulsed signals. They are essentially high-speed ADCs with memory and triggering subsystems, and are often sold in modular form, such as CAMAC and VXI and more recently as plug-in boards. Transient digitizers are like digital oscilloscopes without the display, saving you money when you need more than just a few channels. In fact, the digital oscilloscope as we know it today is somewhat of a latecomer. There were modular digitizers, supported by computers and with analog CRTs for displays, back in the 1960s. Before that, we used analog oscilloscopes with Polaroid cameras, and digitizing tablets to convert the image to ones and zeros. Anyone pine for the "good ol' days"? The fact is, digitizers make lots of sense today because we have virtual instruments, courtesy of LabVIEW. The digitizer is just so much hardware, but VIs make it into an oscilloscope, or a spectrum analyzer, or the world's fastest strip-chart recorder.

If you can't find a transient recorder that meets your requirements, check out the digital oscilloscopes available from the major manufacturers. They've got amazing specifications, capability, and excellent value these days. You might also consider plug-in oscilloscope boards which have the basic functionality of a 'scope but without the user interface. National Instruments offers a line of plug-in digital oscilloscope boards in PCI, PXI, and PCMCIA formats. At this writing, boards are available with sampling rates up to 100 MHz at 8-bit resolution and up to 16 Msamples of onboard memory, featuring analog as well as digital triggering. Some boards are programmed through NI-DAQ, like any other plug-in board, while others are handled by the NI-SCOPE driver, which is IVI-based. In either case, example VIs make it easy to get started and integrate these faster boards into your application.

There are several other major manufacturers of oscilloscope boards that supply LabVIEW drivers. **Gage Applied Sciences** offers a very wide range of models, in ISA and PCI bus format for Windows machines.

**Input characteristics.** High-speed digitizers, including fast oscilloscopes, generally have a  $50\text{-}\Omega$  input impedance. Low-speed models may be high-impedance, and many offer switchable input impedance. AC or DC coupling is also available on most models, but you can always add your own coupling capacitor if the input is DC-coupled only. All modern digitizers have input amplifiers and/or attenuators to adjust the scale factor. Old CAMAC digitizers were fixed at one range, such as 2 V. You have to supply your own wideband amplifiers and attenuators with those.

**Triggering.** Most digitizers rely on a TTL or ECL-level external trigger to start or stop acquisition. Most models have some form of internal, level- or edge-sensitive triggering, much like an oscilloscope. Through software commands, you begin acquisition by arming the digitizer after which you either poll the digitizer to see if it has received a trigger event, or, in the case of most GPIB instruments, wait for an SRQ telling you that it's time to read the data.

Like digital oscilloscopes, you can store **pretrigger** and/or **posttrigger** data. Once the digitizer is armed, its ADC samples at the desired rate and stores data in high-speed memory, which is arranged as a circular buffer. When the trigger occurs, the current memory location is saved, and sampling continues until the desired number of posttrigger samples are acquired. Because data was being stored before the trigger, you can retrieve pretrigger data as well. This is very useful because there may well be important information that arrives before the trigger event in many experiments. The DAQ library does a similar trick with plug-in boards of all types.

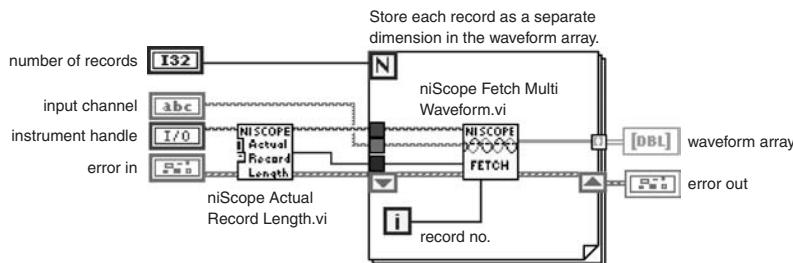
**Data storage and sampling.** An important choice you have to make when picking a transient recorder is how much memory you will need and how that memory should be organized. Single-shot transient events with a well-characterized waveshape are pretty easy to handle. Just multiply the expected recording time by the required number of samples per second. With any luck, someone makes a digitizer with enough memory to do the job. Another option is streaming data to your computer's memory, which is possible with plug-in oscilloscope boards. For instance, a good PCI bus board can theoretically move about 60 million 1-byte samples per second to memory, assuming uninterrupted DMA transfers. Similarly, VXI systems can be configured with gigabytes of shared memory tightly coupled to digitizer modules. It appears that the possibilities with these systems are limited only by your budget.

Far more interesting than simple transient events is the observation of a long tail pulse, such as atomic fluorescence decay. In this case, the intensity appears very suddenly, perhaps in tens of nanoseconds, decays rapidly for a few microseconds, then settles into a quasi-exponential

decay that tails out for the best part of a second. If you want to see the detail of the initial pulse, you need to sample at many megahertz. But maintaining this rate over a full second implies the storage of millions of samples, with much greater temporal resolution than is really necessary. For this reason, hardware designers have come up with digitizers that offer **multiple time bases**. While taking data, the digitizer can vary the sampling rate according to a programmed schedule. For instance, you could use a LeCroy 6810 (which has dual time bases) to sample at 5 MHz for 100 ns, then have it switch to 50 kHz for the remainder of the data record. This conserves memory and disk space and speeds analysis.

Another common experiment involves the recording of many rapid-fire pulses. For instance, the shot-to-shot variation in the output of a pulsed laser might tell you something significant about the stability of the laser's power supplies, flash lamps, and so forth. If the pulse rate is too high, you probably won't have time to upload the data to your computer and recycle the digitizer between pulses. Instead, you can use a digitizer that has **segmented memory**. The Tektronix RTD720A is such a beast. It can store up to 1024 events separated by as little as 5 ns, limited only by the amount of memory installed. After this rapid-fire sequence, you can upload the data at your leisure via GPIB. Other instruments with segmented memory capability are various HP, Tektronix, and LeCroy digital oscilloscopes, and the LeCroy 6810. Figure 19.12 shows how it's done with a National Instruments NI-5112 digital oscilloscope board, which supports **multiple-record** mode, whereby a series of waveforms is stored in onboard memory. The board takes 500 ns to rearm between acquisitions, so you won't lose too much data. This advanced capability is only available through the NI-SCOPE driver.

**Calibration.** As a practical precaution, you should never trust a digitizer when it comes to amplitude calibration. While the time bases are



**Figure 19.12** With the right oscilloscope board and the NI-SCOPE driver, you can rapidly acquire multiple waveform records in onboard memory. This bit of code reads out the data into a 2D array.

generally quite accurate, the same cannot be said for high-speed ADC scale factors. We like to warm up the rack of digitizers, then apply known calibration signals, such as a series of precision dc voltages, and store the measured values. For greater accuracy at high frequencies, you should apply a pulse waveform of known amplitude that simulates the actual signal as closely as possible. If you want to get fancy, add an input switching system that directs all the input channels to a calibration source. Then LabVIEW can do an automatic calibration before and after each experiment. You can either store the calibration data and apply it to the data later or apply the pre-run calibration corrections on the fly. Some plug-in digitizers, like the NI-5112, have onboard calibrators that simplify this task. One precaution: If you have several digitizers with  $50\text{-}\Omega$  inputs, make sure they don't overload the output of your calibrator. It's an embarrassing, but common, mistake.

### Digital storage oscilloscopes (DSOs)

As far as the physics researcher is concerned, the **digital storage oscilloscope (DSO)** has several uses—for troubleshooting and initial setup of experimental apparatus, for convenient display of live waveforms during an experiment, and as a transient digitizer when combined with a LabVIEW system. Most DSOs have a GPIB interface and a LabVIEW driver. Therefore, you can use them to make permanent records of transient events while simultaneously gaining a convenient real-time waveform display. The only limitation we've seen is in some really old models that were incredibly slow at accepting commands and transferring data.

As a utility instrument, the oscilloscope (either analog or digital) is unsurpassed. As much as we would like to think that a digitizer and a LabVIEW display can duplicate an oscilloscope, it never really pans out in the practical world. The reasons are partly aesthetic, partly portability, but mostly that some of us like *real knobs*. Some of the digital oscilloscopes aren't very good in this respect either. All covered with buttons and nary a knob in sight. For general-purpose tests and measurements, we'll take a real 'scope any day.

For real-time waveform display, LabVIEW sometimes approaches the common DSO in terms of display update speed. If you have a modest record length, such as 512 samples, and a fast computer, a GPIB or VXI digitizer can do a credible job of mimicking a real-time display. You can also throw lots of traces on one screen, which is something that most DSOs can't do. Nevertheless, every lab we've ever worked in has had every available oscilloscope fired up, displaying the live waveforms.

## Timing and triggering

The key to your success in pulsed experiments is setting up timing and triggering of all the diagnostic instruments and recording equipment. Just having a rack full of digitizers that all go off at the same time may, or may not, solve your problems. All of your pulsed diagnostics must trigger at the proper time, and you have to know exactly when that trigger occurred.

**What's all this triggering stuff, anyhow?** The experiment itself may be the source of the main trigger event, or the trigger may be externally generated. As an example of an externally triggered experiment, you might trigger a high-voltage pulse generator to produce a plasma. Experiments that run on an internally generated time base and experiments that produce random events, such as nuclear decay, generally are the source of the trigger event. The instruments must be armed and ready to acquire the data when the next event occurs. All of these situations generate what we might classify as a first-level, or primary, trigger (Bologna and Vincelli 1983).

**First-level** triggers are distributed to all of your pulsed diagnostic instruments to promote accurate synchronization. Some utility hardware has been devised over the years to make the job easier. First, you probably need a **trigger fan-out**, which has a single input and multiple outputs to distribute a single trigger pulse to several instruments, such as a bank of digitizers. A fanout module may offer selectable gating (enabling) and inverting of each output. A commercial trigger fanout CAMAC module is the LeCroy 4418 with 16 channels. Second, you often need **trigger delays**. Many times, you need to generate a rapid sequence of events to accomplish an experiment. Modern digital delays are very easy to use. Just supply a TTL or ECL-level trigger pulse, and after a programmed delay, an output pulse of programmable width and polarity occurs. Examples of commercial trigger delays are the Stanford Research Systems DG535 and the LeCroy 4222. Third, you may need a **trigger discriminator** to clean up the raw trigger pulse from your experimental apparatus. It works much like the trigger controls on an oscilloscope, permitting you to select a desired slope and level on the incoming waveform.

A fine point regarding triggering is timing uncertainty, variously known as trigger **skew** or **jitter**. Sometimes you need to be certain as to the exact timing relationship between the trigger event and the sampling clock of the ADC. A situation where this is important is waveform averaging. If the captured waveforms shift excessively in time, then the averaged result will not accurately represent the true wave-shape. Most transient digitizers have a sampling clock that runs all the time. When an external trigger occurs, the current sample number

is remembered by the digitizer's hardware. Note that the trigger event could fall somewhere in between the actual sampling events that are driven by the clock. One way to force synchronization is to use a master clock for your experiment that drives all the digitizers. Then derive the trigger signal from the clock so that triggering always occurs at the same time relative to the clock. Unfortunately, this trick can't be used when the trigger event is random or uncontrollable in nature. In those cases, you must make sure that the trigger generator is stable and repeatable and that the sampling rate is high enough that a timing offset of plus or minus one sample will do no harm.

Figure 19.13 illustrates an application using several of these trigger processing devices with a pulsed laser as the source of the primary trigger event. In this case, the laser excites some plasma phenomenon in a target chamber. A photodiode samples the laser beam to provide a primary trigger pulse. Then a high-voltage pulse generator is triggered to collect ions in the plasma. Finally, the digitizers are triggered to observe the ion current. The delay helps adjust the zero time for the digitizers to center the recorded data with respect to the physical event. Note that a delay generator cannot produce an output that occurs *before* the input. (We can't count the number of times we needed such a device!) If you want a physical event to occur before the trigger, you have to delay everything else in the experiment.

**Second-level triggers** are another useful tool in physics diagnostics. Some experiments require a bit more intelligence to decide when to trigger the data acquisition hardware. For instance, the coincidence of two pulses or digital words (a logical operation) is commonly used to detect high-energy particles. Coincidence detection helps to filter out undesired, anomalous events. In such cases, you can use commercial triggering logic systems, such as those made by LeCroy. Or, if it's a simple problem, just put together some analog and/or digital logic yourself.

**Third-level triggers** represent an even higher level of sophistication in trigger event detection. In many experiments, there may be enough noise or other false events that the first-level and even the second-level

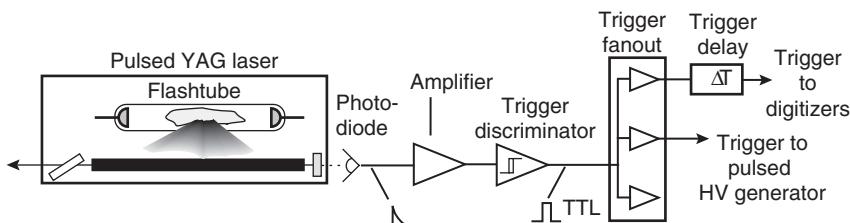


Figure 19.13 Typical application of a trigger discriminator, fanout, and delays used in a pulsed laser experiment.

triggers cause you to record many nonevents. If this occurs very often, you end up with a disk full of empty baselines and just a few records of good data. That's okay, except for experiments where each event consists of 85 megabytes of data. The poor person analyzing the data (you?) is stuck plowing through all this useless information.

The solution is to use a hardware- or software-based, second-level trigger that examines some part of the data before storage. For instance, if you have 150 digitizers, it might make sense to write a subVI that loads the data from one or two of them and checks for the presence of some critical feature—perhaps a critical voltage level or pulse shape can be detected. Then you can decide whether to load and store all the data or just clear the digitizer memories and rearm.

If there is sufficient time between events, the operator might be able to accept or reject the data through visual inspection. By all means, try to automate the process. However, often there is barely time to do any checking at all. The evaluation plus recycle time may cause you to miss the actual event. If there is insufficient time for a software pattern recognition program to work, you can either store all the data (good and bad) or come up with a hardware solution or perhaps a solution that uses fast, dedicated microprocessors to do the pattern recognition.

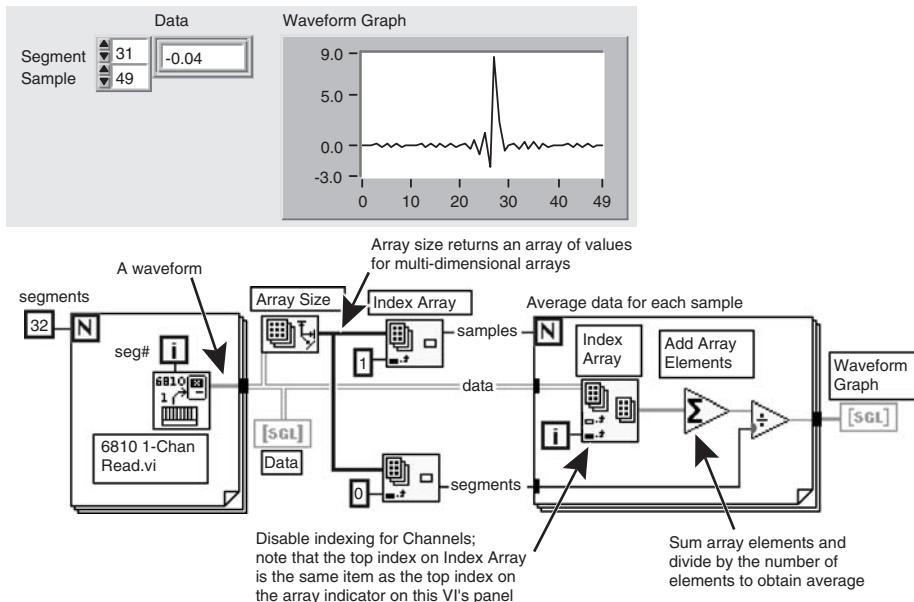
Many modern digital oscilloscopes have a *trigger exclusion* feature that may be of value in physics experiments. You define the shape of a good pulse in terms of amplitude and time, typically with some tolerance on each parameter. After arming, the 'scope samples incoming waveforms and stores any out-of-tolerance pulses. This is good for experiments where most of the data is highly repetitive and nearly identical, but an occasional different event of some significance occurs.

### Capturing many pulses

Many pulsed experiments produce rapid streams of pulses that you need to grab without missing any events. Assuming that you have all the right detectors and wideband amplifiers, you need to select the right acquisition hardware.

There are two ways to treat multipulse data: Capture each pulse individually, or average the pulses as they occur. Averaging has the distinct advantage of improving the signal-to-noise ratio on the order of  $\sqrt{n}$ , where  $n$  is the number of pulses averaged. Of course, if you are able to capture and store all of the pulses, you can always average them afterward.

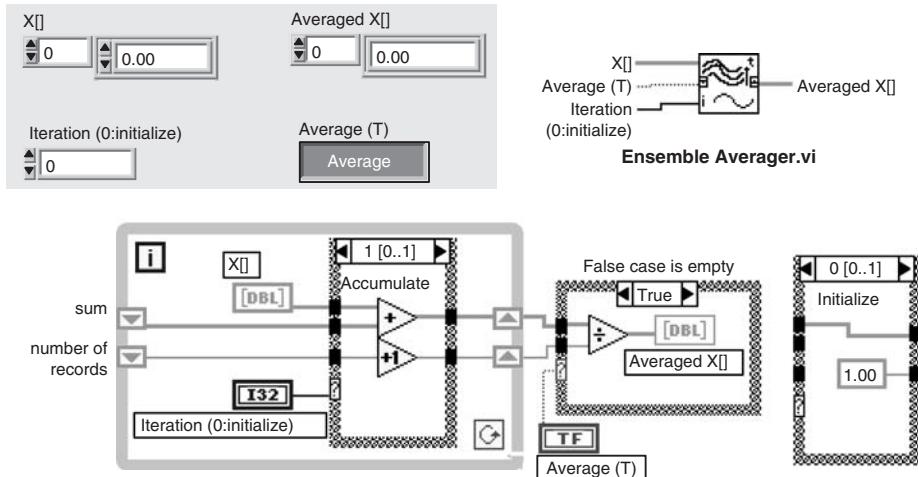
Figure 19.14 is a VI that uses a digitizer with segmented memories to acquire many waveforms. Note that the digitizer must be accurately triggered and in phase with every pulse to avoid statistical errors. Additionally, all the pulses must be similar. The diagram shows some of the array indexing tricks required to average the waveforms.



**Figure 19.14** Averaging waveforms acquired from a digitizer that has segmented memories. Be really careful with array indexing; this example shows that LabVIEW has a consistent pattern with regards to indexes. The first index is always on top, both on the panel and on the diagram.

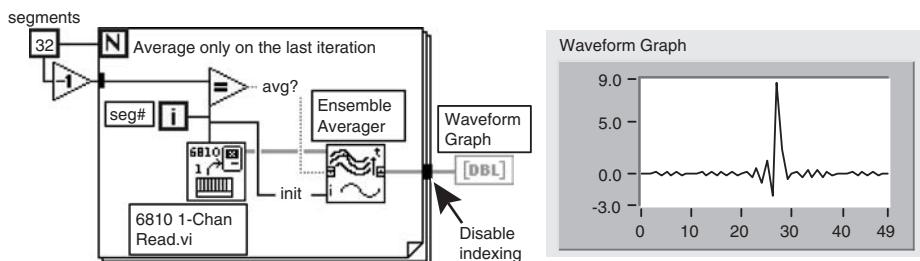
This VI is memory-intensive because it assembles multiple waveforms (1D arrays) into a large 2D array. At first, it may seem that there is no other solution because you need to have all of the waveforms available before averaging can be performed. There is another solution: Use an **ensemble averager** subVI like the one in Figure 19.15. Each time this VI is called, it sums the incoming data with the previous data on a sample-by-sample basis. When called with **Average** set to True, the sum is divided by the number of samples. The **Iteration** control initializes the two uninitialized shift registers. At initialization time, the incoming waveform determines the length of the array.

Now you can rewrite the data-averaging VI in a way that saves memory. The improved version is shown in Figure 19.16, and it produces the same result but with much less memory usage. To further improve performance, averaging is turned off until the last iteration of the For Loop. This step saves many floating-point division operations. If you want to see the averaging operation evolve in real time, you could move the Waveform Graph inside the For Loop and turn averaging on all the time. Each time a new segment of data is read, the graph will be updated with a newly computed average.



**Figure 19.15** This shows an ensemble averager VI. It accumulates (sums) the data from multiple waveforms, then calculates the average on command. It uses uninitialized shift registers to store the data between calls.

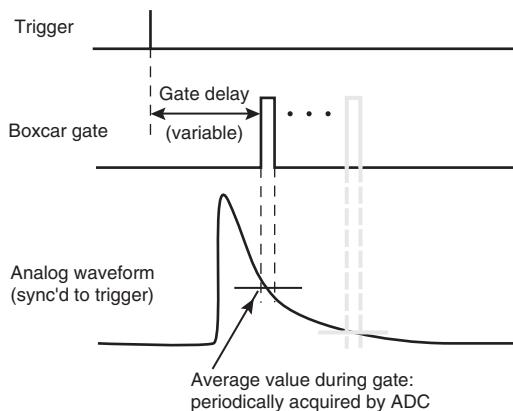
Individual pulses need to be stored when the information contained in each pulse is significant. The big issues (as far as LabVIEW is concerned) are how often the pulses occur and how fast your data transfer rate and throughput are. If the pulses only occur every second or so, there is no problem. Any simple acquisition loop that you write with a good external digitizer will probably work. But sometimes you will want to capture pulses at much higher rates, perhaps hundreds or thousands of pulses per seconds. That calls for digitizers with segmented memories, as previously mentioned. For intermediate rates, it's *possible* to do the acquisition in LabVIEW, but you had better do some testing to verify performance. Remember to optimize your program (described later in this chapter), primarily by minimizing graphics and memory usage.



**Figure 19.16** Using the ensemble averager drastically simplifies the diagram and cuts memory usage while yielding the same results as before.

If the running average value of many pulses is interesting, then you can use some other instruments. Most DSOs offer built-in signal averaging. Assuming that you can trigger the DSO reliably, this is a really easy way to obtain averaged waveforms over GPIB. For a more modular approach, DSP Technology makes a line of averaging memories for use with its CAMAC digitizers. It uses high-speed memory and an adder to accumulate data from synchronized waveforms. After the desired number of events ( $N$ ), you read out the memory and divide the data by  $N$ . Some advanced VXI modules may support this kind of waveform acquisition and processing as well.

You can also use a **boxcar averager** to acquire averaged pulse waveforms. The basic timing diagram for a boxcar averager is shown in Figure 19.17. The gate interval is a short period of time during which the analog signal voltage is sampled and its amplitude stored as charge on a capacitor. The output of the boxcar is the average of many such gate events over a selected period of time. If the gate is fixed at one location in time, relative to the recurrent analog waveform, then the boxcar output represents the average value of the waveform at that time—just like putting cursors on the graph of the waveform. Now we can vary the gate delay with an analog voltage, thus scanning the sampling location over the entire period of the pulse. By scanning slowly, the boxcar output accurately reconstructs the analog waveform. With this technique, a simple *xy* chart recorder or slow data acquisition system can record the shape of *nanosecond* waveforms.



**Figure 19.17** Basic timing diagram for a boxcar averager. The output of the boxcar is the average of many measurements that are acquired during many gate events. By slowly varying the gate delay, the boxcar output reconstructs the shape of the analog waveform.

One boxcar system, the Stanford Research Systems SR250, offers a GPIB interface module (for which we have a LabVIEW driver) that directly supports gate scanning and also measures the boxcar output. Even without this special support, you can use a ramp generator VI like the one described in the section on Plasma Potential Experiments to scan the gate and make the output measurements. On the SR250, the gate delay signal is a 0–10-V input (drive it directly from your MIO-16 output), and the gate delay ranges from 1 ns to 10 ms, so you can cover a very wide range of pulse widths. It also offers a high-speed sampling module that shrinks the gate time to about 10 ps. The boxcar output ranges up to  $\pm 2$  V and is well filtered, making accurate measurements a snap.

**Equivalent time sampling (ETS)** is a trick used by high-speed digitizers, DSOs, and the E-series DAQ and oscilloscope boards from National Instruments. Realize that very fast ADCs are expensive and often lack high resolution. But sample-and-hold amplifiers can be designed with very short aperture (sampling) times—on the order of picoseconds—at a more reasonable cost. This allows the hardware designer to use a scheme similar to that of the boxcar averager. They delay the triggering of the sample-and-hold and ADC by a variable amount, just like the boxcar’s gate, thus scanning the sampling point across the period of a recurrent waveform. Once the sample-and-hold has acquired a voltage, a low-speed ADC can read that voltage at a relatively leisurely rate and with good resolution. The trigger delay can be chosen randomly or it may proceed in a monotonic fashion like the boxcar. Thus, the waveform is reconstructed in memory for readout and display.

ETS, like the boxcar, is a specialized *hardware* solution that solves a particular problem: acquiring fast, recurrent waveforms. Physics diagnostic problems often require esoteric hardware like that, and you should try to learn about special instruments to enhance your arsenal of solutions. LabVIEW alone can’t replace every kind of system, but it can sure make the job easier, once you’ve got the right front-end hardware.

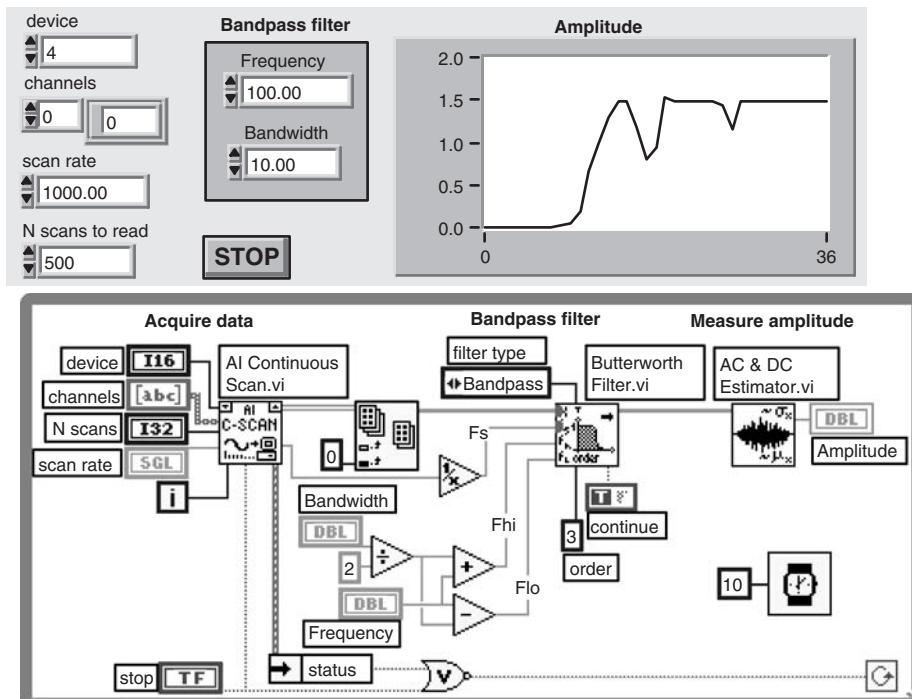
### Recovering signals from synchronous experiments

Say that you have a beam of light chopped at 10 kHz that is partially absorbed in an experiment. The remaining light strikes a photodiode, producing a small-amplitude square wave, and you want to measure the amplitude. To complicate things, assume the likely scenario where there is a great deal of 60-Hz noise and some nasty impulse noise from other apparatus. If you digitize this signal and display it, there’s a good chance you won’t even *find* the 10-kHz waveform, let alone measure its amplitude. You must do some signal processing first.

One solution is to apply a narrow-bandwidth digital bandpass filter, tuned to the desired center frequency. The concept is also known as a **frequency-selective voltmeter**. It's simple and often adequate because much out-of-band noise is rejected. All it takes is a DAQ waveform acquisition VI, followed by one of the IIR filter VIs. To determine the amplitude, you might use the **AC & DC Estimator** VI from the Measurement function palette in the Analysis library. Its ac amplitude estimate is obtained from the mean and variance of the time domain signal after passing the signal through a Hann time window.

In the lock-in library you'll find a VI called **Signal Recovery By BP Filter**. It acquires data from one channel of a DAQ board, applies a low-pass filter, and displays the ac estimate (Figure 19.18).

There are some shortcomings to this filter-based technique. Impulses and other transient signals often contain energy that falls within the passband of the filter, making that polluting energy inseparable from your signal. Also, filters with narrow passbands (high  $Q$ ) are prone to ringing when excited by impulse or step waveforms. If the incoming signal is sufficiently noisy, you may not be able to recover your



**Figure 19.18** The Signal Recovery by BP Filter VI demonstrates a way of measuring amplitude in a selected frequency band. For this screen shot, I swept the input frequency around the 100-Hz region while running the VI.

signal reliably. To see how bad the problem can be, try the Pulse Demo VI from the Analysis examples. Turn the noise way up, and watch the pulse disappear from view. Note that the addition of some signal conditioning filters and amplifiers, plus proper grounding and shielding, may remove some of the interference from your signal and is always a worthwhile investment.

In experiments that operate with a constant carrier or chopping frequency, the signal amplitude can be recovered with **lock-in amplifiers**, which are also called **synchronous amplifiers** or **phase-sensitive detectors (PSDs)**. The basic principle revolves around the fact that you need only observe (demodulate) the range of frequencies within a narrow band around the carrier frequency. To do this, the lock-in requires a reference signal—a continuous sine or square wave that is perfectly synchronized with the excitation frequency for the experiment. The lock-in mixes (multiplies) the signal with the reference, resulting in an output containing the sum and difference frequencies. Since the two frequencies are equal, the difference is zero, or dc. By low-pass-filtering the mixer output, you can reject all out-of-band signals. The bandwidth, and hence the response time, of the lock-in is determined by the bandwidth of the low-pass filter. For instance, a 1-Hz lowpass filter applied to our example with the 10-kHz signal results in (approximately) a 1-Hz bandpass centered at 10 kHz. That's a very sharp filter, indeed, and the resulting signal-to-noise ratio can be shown to be superior to any filter (Meade 1983).

Lock-ins can be implemented by analog or digital (DSP) techniques. It's easiest to use a commercial lock-in with a LabVIEW driver, available from Stanford Research or EG&G Princeton Applied Research. If you want to see a fairly complete implementation of a lock-in in LabVIEW, check out the example in the lock-in library. For a simple no-hardware demonstration, we created a two-phase lock-in with an internal sinusoidal reference generator. This example VI, **Two-phase Lock-in Demo**, simulates a noisy 8-kHz sine wave, and you can change the noise level to see the effect on the resulting signal-to-noise ratio.

The **DAQ 2-Phase Lock-in VI** is a real, working, two-phase DSP lock-in where the signal and reference are acquired by a plug-in board. The reference signal can be a sine or square wave, and the amplitude is unimportant. SubVIs clean up the reference signal and return a quadrature pair of constant-amplitude sine waves. To generate these quadrature sine waves, we had to implement a phase-locked loop (PLL) in LabVIEW. We used a calibrated IIR low-pass filter as a phase shift element. When the filter's cutoff frequency is properly adjusted, the input and output of the filter are exactly 90 degrees out of phase. This condition is monitored by a phase comparator, which feeds an error signal back to the low-pass filter to keep the loop in lock.

We tested this application on a Macintosh G3/400 with a PCI-MIO-16E1 board. It's possible to obtain stable amplitude and phase measurements with sampling rates up to at least 30 kHz, processing buffers as short as 500 samples. That yields a maximum demodulated (output) bandwidth of up to 30 Hz. A 1-kHz signal can be accurately measured when its rms amplitude is below 1 LSB on the ADC if the time constant is long enough. To make this application run faster, you can eliminate graphical displays or get a faster computer.

## Handling Huge Data Sets

Experiments have a way of getting out of hand with respect to the quantity of data generated. At the NASA Ames Research Center's Hypervelocity Free-Flight Aerodynamic Facility (HFFAF) (Bogdanoff et al. 1992), they use about 124 channels of transient digitizers with anywhere from 8 to 256K samples per channel—about 16 megabytes per shot, typical. (The good news is, they can only do a couple of shots per week.) In another common situation, seismology, they need to digitize and record data at hundreds of samples per second from many channels—forever. *Forever* implies a *really big* disk drive. Obviously, there must be some practical solutions to memory and disk limitations, and that's what we'll look at next.

### Reducing the amount of data

The concept of third-level triggers can certainly reduce the amount of data to be handled. Or, you can do some kind of preliminary data reduction to compact the raw data before storing it on disk.

In seismology, an algorithm called a *P-picker* continuously processes the incoming data stream, searching for features that indicate the presence of a P wave (a high-frequency, perpendicular shock wave), which indicates that an interesting seismic event has occurred. If a P wave is detected, then a specified range of recent and future data is streamed to disk for later, more detailed analysis. Saving all of the raw data would be impractical, though they sometimes set up large circular buffer schemes just in case a really important event (“*The Big One*”) occurs. That permits a moderate time history to remain in memory and/or on disk for immediate access. But in the long run, the P-picker is key.

If your data is of a contiguous but sparse nature, like that of a seismic data recording system, consider writing a continuous algorithm to detect and keep interesting events. Because the detection algorithm may require significant execution time, it probably needs to run asynchronously with the data acquisition task. If you are using a plug-in board, you can use buffered DMA. Some kinds of smart data acquisition units also support

buffering of data, such as the HP3852. Either way, the hardware continues collecting and storing data to a new buffer even while the computer is busy analyzing the previous buffer.

If you can't afford to throw away any of your data, then data reduction on the fly might help reduce the amount of disk space required. It will also save time in the post-experiment data reduction phase. Is your data of a simple, statistical nature? Then maybe all you really need to save are those few statistical values, rather than the entire data arrays. Maybe there are other characteristics—pulse parameters, amplitude at a specific frequency, or whatever—that you can save in lieu of raw data. Can many buffers of data be averaged? That reduces the stored volume in proportion to the number averaged. If nothing else, you might get away with storing the reduced data for every waveform, plus the raw data for every *N*th waveform. Decimation is another alternative if you can pick a decimation algorithm that keeps the critical information, such as the min/max values over a time period. As you can see, it's worth putting some thought into real-time analysis. Again, there may be performance limitations that put some constraints on how much real-time work you can do. Be sure to test and benchmark before committing yourself to a certain processing scheme.

### Optimizing VIs for memory usage

Large data sets tend to consume vast quantities of memory. Some experiments require the use of digitizers that can internally store millions of samples because there simply isn't time during the run to stream the data to your computer. When the run is over, it's time to upload, analyze, and save all those samples. If your computer has only, say, 32 MB of RAM, then it's unlikely that you will be able to load more than a few hundred thousand samples of data in one chunk. You have two choices: Buy more memory or write a smarter program.

Expanding the memory capacity of your machine is certainly worthwhile. LabVIEW is a big memory user, no doubt about it. And modern operating systems aren't getting any smaller. Even **virtual memory (VM)**, available on all of LabVIEW's platforms, may not save you because there are significant real-time performance penalties due to the disk swapping that VM causes. For best performance and simplicity in programming, nothing beats having extra memory. Of course it costs money, but memory is cheap compared to the aggravation of poor system performance.

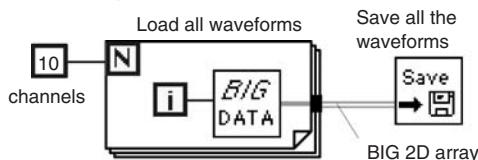
So, you find yourself a little short on memory, a little short on cash, and a little down in the mouth because LabVIEW dies every time you try to load those megasamples of data. It's time to optimize, and perhaps rewrite, your program. By the way, we guarantee that you

will see an increase in the *performance* of your program when you optimize memory usage. Anything you do to reduce the activity of our old friend the memory manager will speed things up considerably. Here are the basic memory optimization techniques that everyone should know.

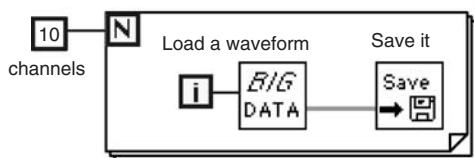
Begin by “RTFMing” (reading the fine manual), this time concentrating on the chapter in the LabVIEW user manual called “Performance Issues.” The LabVIEW team is fully aware of the importance of explaining how LabVIEW allocates memory, and they continue to improve and expand the available documentation. Also check the current technical notes. LabVIEW Application Note AN168, ***LabVIEW Performance and Memory Management***, is relevant to the current problem: handling large quantities of data.

**Tip 1: Use cyclic processing rather than trying to load all channels at once.** Once you see this concept, it becomes obvious. As an example, here is a trivial example of the way you can restructure a data acquisition program. In Figure 19.19A, 10 waveforms are acquired in a For Loop, which builds a 10-by- $N$  2D numeric array. After collection, the 2D array is passed to a subVI for storage on disk. The 2D array is likely to be a memory burner. Figure 19.19B shows a simple, but effective change. By putting the data storage subVI in the loop, only one 1D array of data needs to be created, and its data space in memory is reused by each waveform. It’s an instant factor-of-10 reduction in memory usage.

**A.** All the data is loaded at once into a huge 2D array.  
Not memory efficient!



**B.** One waveform at a time is loaded and written to a file. Much more memory-efficient.



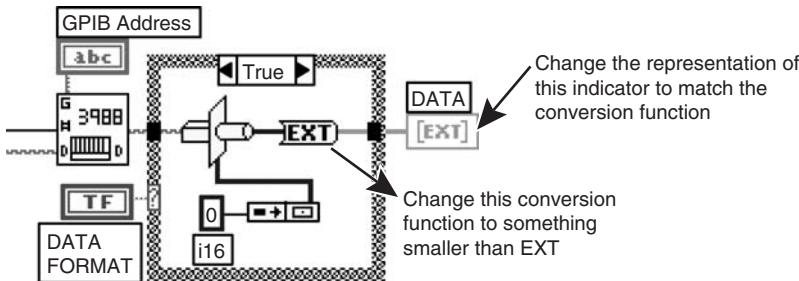
**Figure 19.19** Use cyclic processing of data (b) to avoid storing all of your data in memory at the same time, as in a.

Of course, you can't always use cyclic processing. If you need to perform an operation on two or more of the waveforms, they have to be concurrently memory-resident. But keep trying: If you only need to have *two* waveforms in memory, then just keep *two*, and not the whole bunch. Perhaps you can use shift registers for temporary storage and some logic to decide which waveforms to keep temporarily.

**Tip 2: Break the data into smaller chunks.** Just because the data source contains a zillion bytes of data, you don't necessarily have to load it all at once. Many instruments permit you to read selected portions of memory, so you can sequence through it in smaller chunks that are easier to handle. The LeCroy 6810 CAMAC digitizer and most digital oscilloscopes have such a feature, which is supported by the LabVIEW driver. If you're using a plug-in board, the DAQ VI **AI Read** permits you to read any number of samples from any location in the acquisition buffer. Regardless of the data source, it's relatively simple to write a looping program that loads these subdivided buffers and appends them to a binary file one at a time. Similarly, you can stream data to disk and then read it back in for analysis in manageable chunks.

**Tip 3: Use the smallest practical numeric types.** Remember that LabVIEW supports many numeric types, from a single-byte integer on up to a 16-byte extended-precision (EXT) float. The minimum amount of memory that your data will consume is the number of samples multiplied by the size of the numeric type, in bytes. Lots of digitizers have 8-bit resolution, which fits perfectly into a single-byte integer, either signed (I8) or unsigned (U8). That's a case of maximum efficiency. Many times, however, you will want to scale the raw binary data to engineering units which require a floating-point representation. For most cases, try to use the single-precision float (SGL), which is 4 bytes long and offers seven significant figures of precision. In the rare case that greater precision is required, use double-precision (DBL) which occupies 8 bytes. The latter type is also required by all VIs in the Analysis library. If you use any of those VIs, you may as well keep all of your data in DBL format. Why? That's the next tip.

**Tip 4: Avoid coercion and conversion when handling arrays.** Minimizing data space requires constant vigilance. Begin by looking at your data sources, such as driver VIs. Make sure that the data is born in the smallest practical type. Drivers that were created way back in version 1.2 of LabVIEW are notorious for supplying all data in EXT format (the only numeric type then available). Figure 19.20 shows one such example, an old CAMAC driver VI. You might even create your own versions of driver VIs, optimized for your memory-intensive application.



**Figure 19.20** This old driver (from the Kinetic Systems 3988 library) converts 16-bit binary data to EXT floating point, which wastes much memory. Change the indicated conversion function to something smaller, such as SGL. Save big! And get the Ginsu knife!

Most of the analysis functions in the LabVIEW library are partly polymorphic and handle only DBL-format floating-point or waveform data. Using one of these functions on your more compact SGL data results in a doubling of memory usage. There are several things you can do if you need to conserve memory. First, call National Instruments and complain that you want totally polymorphic analysis VIs. Second, if it's not too complex, try writing your own equivalent function subVI in LabVIEW, one which uses only your chosen (smaller) data type. Realize that your routine may run a bit slower than the optimized, CIN-based analysis VI. The **G Math Toolkit** contains a plethora of analysis VIs that use no CINs, and that can be the basis for your custom, memory-optimized analysis. Third, you can perform the analysis at a time when less data is resident in memory—perhaps after the data acquisition phase is complete.

Keep a sharp eye open for **coercion dots** at terminals that handle large arrays. Anywhere you see one, the data type is being changed, and that means that reallocation of memory is required. This takes time and, especially if the new data type is larger, results in the consumption of even more memory. If the coercion dot appears on a subVI, open the subVI and try to change all of the related controls, indicators, and functions to the same data type as your source. Eliminate as many coercion dots as you can. Sometimes this is impossible, as with the Analysis library, or with any VI that contains a CIN. By the way, *never* change the data type that feeds data into a CIN! A crash will surely ensue because the CIN expects a certain data type; it has to be recompiled to change the data type. Maybe one day we'll have polymorphic CINs, too. Finally, don't convert between data types unnecessarily. For instance, changing a numeric array to a string for display purposes is costly in terms of memory and performance. If you must display data, try to use an indicator in the native format of that data.

**Tip 5: Avoid data duplication.** LabVIEW Application Note AN168, *LabVIEW Performance and Memory Management*, goes into some detail on the subject of efficient use of memory. In general, the G compiler tries very hard to avoid data duplication by reusing memory buffers. You can help by choosing the best combination of functions and structures to promote reuse. Here are some of the most important rules to keep in mind.

- *An output of a function reuses an input buffer if the output and the input have the same data type, representation, and—for arrays, strings, and clusters—the same structure and number of elements.* This category includes all of the Trig & Log functions; most of the Arithmetic functions, including those that operate on boolean data; a few string functions such as *To Upper Case*; the Transpose Array function; and the Bundle function to the extent that the cluster input and output can use the same buffer. Most other functions do not meet this criteria; among them are the Build Array and Concatenate String functions.
- *In general, only one destination can use the buffer of the data source.* More specifically, where the source of a data path branches to two or more destinations that *change* the input data, the source buffer can be used by only one. A way to conserve memory is to do processing in a serial fashion—one subVI after another—rather than wiring the subVIs in parallel.
- *If there are multiple destinations that read data but do not change it, they may use the same source buffer.* For example, Array Size doesn't duplicate the buffer since it doesn't change it. Also, under certain circumstances, when a node that alters data and one or more nodes that merely read data have the same source, LabVIEW can determine that the readers execute before the node that changes the data so that the source buffer can be reused. (For this to occur, all nodes involved must be synchronous.)
- *If the source is a terminal of a control that is not assigned to a connector terminal or receives no data from the calling VI, the source data is stored in a separate buffer.* Without an external source to supply data when the subVI executes, the data is locally supplied. This means you can save memory by not trying to display it. You can also use the VI Properties dialog to change the priority to **Subroutine**. Subroutine priority eliminates all extra buffers associated with indicators and slightly reduces subVI calling overhead. Note that it also prevents you from running the VI interactively!
- *If a buffer can be reused by an input and output of a structure or subVI, the buffer can be used by other nodes wired to that node.* Thus, in a

hierarchical program, a buffer can be reused from the top-level VI to the lowest subVI and back up to the top. Don't be afraid to use subVIs, so long as they contain no additional array duplications internally.

- *Data allocated to a top-level VI cannot be deallocated during execution.*  
On the other hand, data allocated in a subVI can be deallocated.

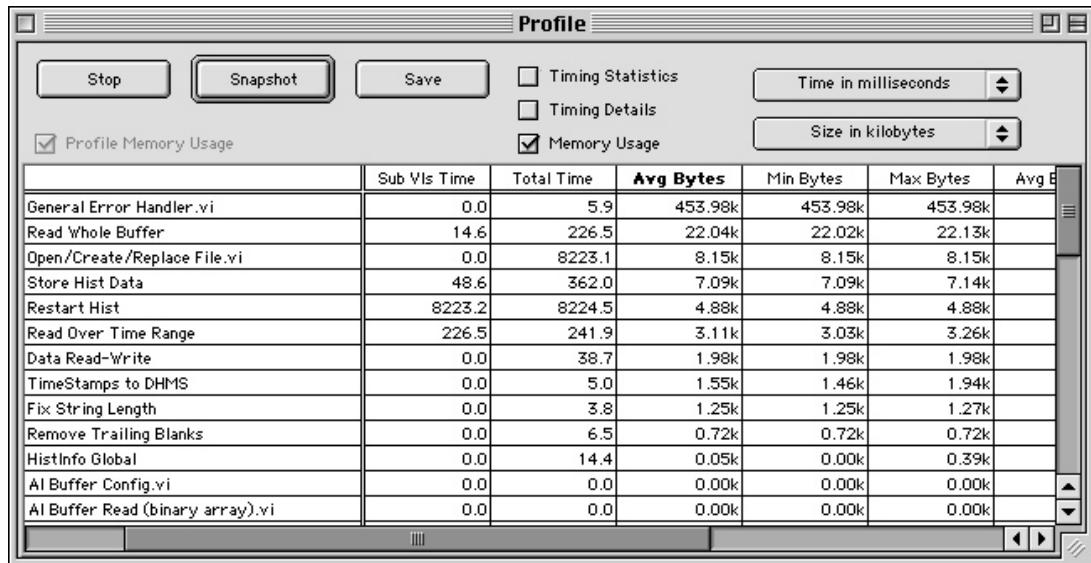
You have some control over the LabVIEW memory manager through the Options dialog, in the **Performance and Disk** items. The first option is *Deallocate memory as soon as possible*. When a subVI finishes execution, this option determines whether or not its local buffers will be deallocated immediately. If they are deallocated, you gain some free memory, but performance may suffer because the memory will have to be *reallocated* the next time the subVI is called. On the Macintosh, there is an option called *Compact memory during execution*. When enabled, LabVIEW tells the Macintosh memory manager to defragment memory every 30 seconds. Defragmented memory enhances speed when allocating new buffers. However, this compacting operation usually induces a lull in performance every 30 seconds which may be objectionable in real-time applications.

If your application uses much more memory than you think it should, it's probably because of array duplication. Try to sift through these rules (study the application note for more) and figure out where you might modify your program to conserve memory.

**Tip 6: Use memory diagnostics.** There are some ways to find out how much memory your VIs are actually using. Please note that these techniques are not absolutely accurate. The reasons for the inaccuracy are generally related to the intrusiveness of the diagnostic code and the complexity of the memory management system. But as a relative measure, they are quite valuable.

Begin by using the **VI Properties** dialog in the File menu to find out how much memory is allocated to your VI's panel, diagram, code, and data segments. Of primary interest is the data segment, because that's where your buffers appear. Immediately after compiling a VI, the data segment is at its minimum. Run your VI, and open the VI Properties dialog again to see how much data is now allocated. You can try some simple tests to see how this works. Using an array control, empty the array and check the data size. Next, set the array index to a large number (such as 10,000), enter a value, and check the data size again. You should see it grow by 10,000 times the number of bytes in the data type.

The most extensive memory statistics are available through the **Profiling** feature of LabVIEW. Select Profile VIs from the Tools >> Advanced menu, and start profiling with the **Profile Memory Usage** and **Memory Usage** items selected (Figure 19.21). Run your VIs, take



**Figure 19.21** The Profiling window gives you timing and memory usage statistics on your application. The first VI in the list is the biggest memory user; that's the place to concentrate your efforts.

a snapshot at some critical time, and examine the memory-related columns. If you click on a column header, the entire spreadsheet will be sorted in descending order by that column. Important statistics include minimum, maximum, and average bytes used, in addition to timing information. Find out which VIs use the most memory and try to optimize them first. The Profiler is a quantitative tool for this effort. Sometimes, the least-expected VI is the biggest memory hog!

## Bibliography

- Accelerator Technology Division AT-5: *Low-Level RF LabVIEW Control Software User's Manual*, Los Alamos National Laboratory, LA-12409-M, 1992. (Available from National Technical Information Service, Springfield, Va.)
- Bogdanoff, D. W., et al.: "Reactivation and Upgrade of the NASA Ames 16 Inch Shock Tunnel: Status Report," *American Institute of Aeronautics and Astronautics* 92-0327.
- Bologna, G., and M. I. Vincelli (Eds.): "Data Acquisition in High-Energy Physics," *Proc. International School of Physics*, North-Holland, Amsterdam, 1983.
- Mass, H. S. W., and Keith A. Brueckner (Eds.): *Plasma Diagnostic Techniques*, Academic Press, New York, 1965.
- Meade, M. L.: *Lock-in Amplifiers: Principles and Applications*, Peter Peregrinus, Ltd., England, 1983. (Out of print.)
- 2000 LeCroy Research Instrumentation Catalog, LeCroy Corporation, New York, 2000. ([www.lecroy.com](http://www.lecroy.com).)

*This page intentionally left blank*

## Data Visualization, Imaging, and Sound

Today's sophisticated computer systems make it possible for you to collect and observe information in ways that used to be impractical or impossible. Once the bastion of the High Priests of Graphic Arts, the ability to produce multidimensional color plots, acquire and manipulate images, and even make movies of your data is directly available (live!) through LabVIEW. All you need to do is configure your system appropriately and, for some features, buy some third-party LabVIEW applications that offer these advanced capabilities. This chapter tells you how.

We humans have a marvelous ability to absorb visual information, but only when it is presented in optimum form. A well-designed graph or image conveys a great deal of information and gives us unexpected insight into hidden relationships buried in our data. You have to invest some programming effort, time, money, and thought to create the right visualization instrument, but the results are both valuable and impressive.

Keep in mind the fact that **imaging** is just another data acquisition and analysis technique. That's a fact that is overlooked when you haven't seen it in action. Images can be born of cameras and live scenes—the usual pictures—or they can arise from other forms of numerical data. Images are more than pretty pictures; they can be quantitative as well, spatially and in intensity. Further, you can make those images into movies and even combine them with sound by using the new concepts embodied in **multimedia**. LabVIEW does all of these things with the help of some add-on products.

There are dozens, perhaps hundreds, of data visualization (graphing, imaging, and presentation) programs available on various computer platforms these days. But LabVIEW has one big advantage over most

of them: It can tie in data visualization with other operations in your real-world laboratory. For example, you can direct a robot arm, measure temperatures, control an electron beam welder, acquire images of the weld, process and display those images, store data on disk . . . and do it *all in real time, right there in LabVIEW*. This is an exceptional capability! Only a few years ago, you had to use several different programs (probably not simultaneously, either) or write a zillion lines of custom code to perform all of these tasks. Sometimes, the processing you wish to perform isn't practical in LabVIEW. In that case, you can still exchange data files with other specialized programs on various computers.

This chapter covers graphing of data, image acquisition and processing, and the recording and production of sound. LabVIEW has built-in graphs that are adequate for both real-time and posttest data presentation of ordinary two-dimensional data. It also has the capability to display images in color or gray scale. With the addition of add-on products discussed here, you can also acquire and analyze video images, make multidimensional plots, QuickTime movies, and other advanced displays.

## Graphing

The simplest way to display lots of data is in the form of a graph. We've been drawing graphs for about 200 years with pens and paper, but LabVIEW makes graphs faster and more accurate. Most important, they become an integral part of your data acquisition and analysis system.

Part of your responsibility as a programmer and data analyst is remembering that a well-designed graph is intuitive in its meaning and concise in its presentation. In his book *The Visual Display of Quantitative Information* (1983), Edward Tufte explains the fundamentals of graphical excellence and integrity and preaches the avoidance of graphical excess that tends to hide the data. We've been appalled by the way modern presentation packages push glitzy graphics for their own sake. Have you ever looked at one of those 256-color three-dimensional vibrating charts and tried to *actually see the information?*

Here are your major objectives in plotting data in graphical format:

- Induce the viewer to think about the substance rather than the methodology, graphic design, the technology of graphic production, or something else.
- Avoid distorting what the data has to say (for instance, by truncating the scales, or plotting linear data on logarithmic scales).

- Try to present many numbers in a small space, as opposed to diffusing just a few data points over a vast area. This makes trends in the data stand out.
- Make large data sets more understandable by preprocessing.
- Encourage the eye to compare different pieces of data.
- Reveal the data at several levels of detail, from a broad overview to the fine structure. You can do this by making more than one plot with different scales or by allowing the user to rescale a graph—a standard LabVIEW feature.
- Design the graph in such a way that it serves a clear purpose with respect to the data—for description, exploration, or tabulation.
- The graph should be closely integrated with other descriptions of the data. For example, a graph should be synergistic with numeric displays of statistics derived from the same data.
- Note that some data is better displayed in a format other than a graph, such as a table.
- Above all, make sure that the graph *shows the data*.

The worst thing you can do is over decorate a graph with what Tufte calls *chartjunk*. Chartjunk is graphical elements that may catch the eye but tend to hide the data. You don't want to end up with a graphical puzzle. Simply stated, *less is more* (Figure 20.1). Though this is more of a problem in presentation applications, there are some things to avoid when you set up LabVIEW graphs:

- High-density grid lines—they cause excessive clutter.
- Oversized symbols for data points, particularly when the symbols tend to overlap.
- Colors for lines and symbols that contrast poorly with the background.
- Color combinations that a color-blind user can't interpret (know thy users!).
- Too many curves on one graph.
- Insufficient numeric resolution on axis scales. Sometimes scientific or engineering notation helps (but sometimes it hinders—the number 10 is easier to read than 1.0E1).
- Remove or hide extra bounding boxes, borders, and outrageous colors. Use the coloring tool, the transparent (T) color, and the control editor to simplify your graphs.



...but maybe all you really need is the data itself. Axis values are only required for quantitative graphing. Controls and legends should not be displayed unless needed.

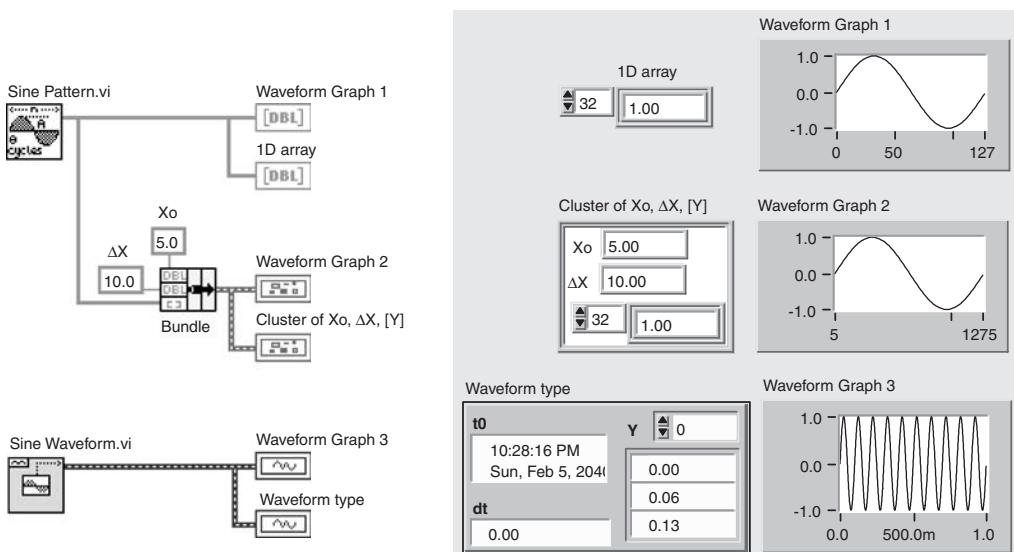


**Figure 20.1** Sometimes, less is more when presenting data in graphs. Use the coloring tool, the control editor, and the various pop-up menus to customize your displays.

### Displaying waveform and cartesian data

Most data that you'll acquire is a function of time or some other variable and is described by  $y = f(x)$ , where  $x$  is the independent variable (e.g., time) and  $y$  is the dependent variable (e.g., voltage). This is generalized **cartesian** ( $xy$ ) data. The  $x$  values are most often based on time. Time-series, or **waveform**, data is effectively displayed as an ordinary graph or, in the case of real-time data, as a strip chart. Graphs and charts are standard LabVIEW indicators with many programmable features. Read through the graphing chapter of the LabVIEW user manual to see how all those features are used. In particular, Property Nodes are worthy of much study because graphs have a large number of varying features.

Figure 20.2 shows the simplest ways to use the **Waveform Graph**. Like many LabVIEW indicators, it's polymorphic, so you can wire several data types directly to it. Numeric arrays (any numeric representation) can be wired directly and are displayed with the  $x$  axis scaled with values beginning with zero ( $x_0 = 0$ ) and increasing by one ( $\Delta x = 1$ ) for each data point in the array. You can also control  $x_0$  and  $\Delta x$  by bundling your data into a cluster as shown in Figure 20.2. Finally, the waveform data type can be wired directly to a Waveform Graph, and all the scaling information is carried along. Any time your data has evenly spaced  $x$  values, use the Waveform Graph. Compared with an XY Graph, it takes less memory, is faster, and is certainly the easiest to set up.

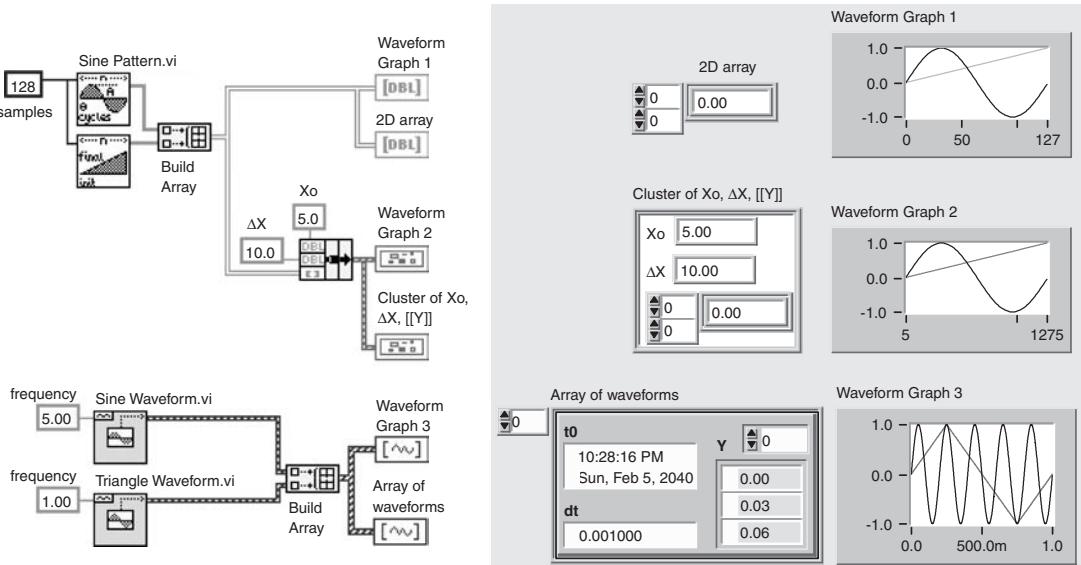


**Figure 20.2** The Waveform Graph is polymorphic. This example shows some of the basic data types that it accepts for the making of single-variable plots.

**Polymorphism** makes the Waveform Graph more versatile because the graph directly accepts several data types. In the previous example, you saw that a single variable could be plotted with or without *x*-axis scaling. Figure 20.3 shows how to make a multiplot graph where each signal has the same number of points. You scale the *x* axis in a manner consistent with the previous example.

What if the signals don't have the same number of points? Two dimensional arrays require that each row of data has the same number of points, so that approach won't work. Your first guess might be to make an array of arrays. Sorry, can't do that in LabVIEW. Instead, make an array that contains a cluster with an array inside it (Figure 20.4). There are two ways to build this data structure. One uses a **Bundle** function for each array followed by **Build Array**. The other way uses the **Build Cluster Array** function, which saves some wiring. The index of the outer array is the plot number, and the index of the inner array is the sample number.

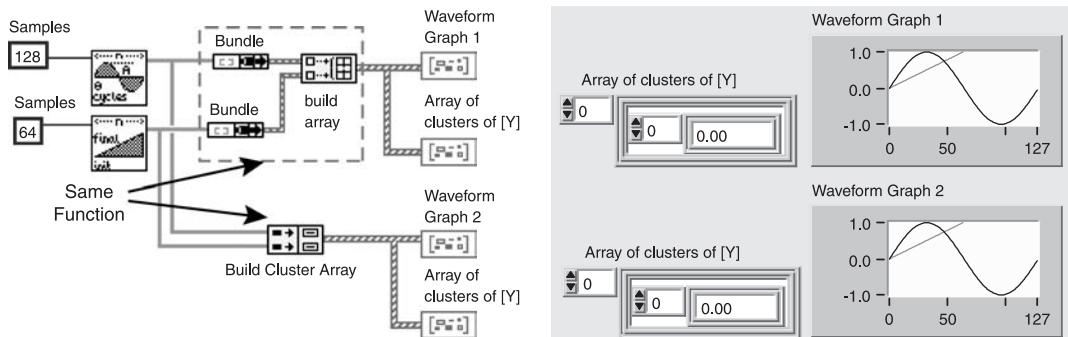
A slightly different approach, shown in Figure 20.5, is required if you want to make multiple plots with different numbers of points *and* you want to scale the *x* axis. For this example, we plotted a sine wave and a line representing its mean value (which might be useful for a control chart in statistical process control). The mean value is used as both elements of a two-element array, which is the *y*-axis data for the



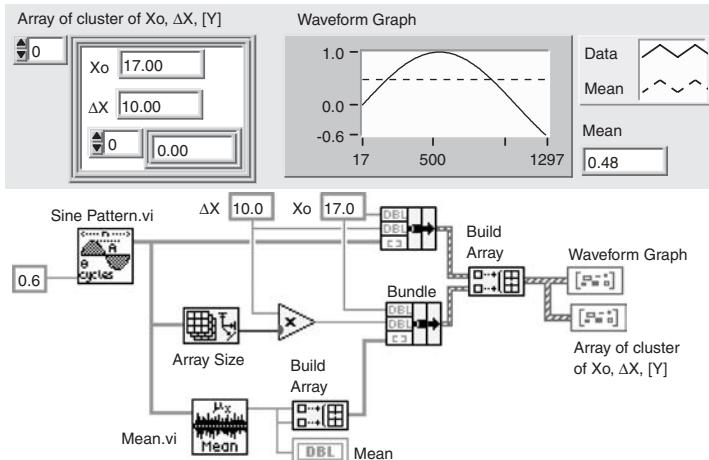
**Figure 20.3** Three ways of making multiple plots with a Waveform Graph when the number of data points in each plot is equal. Note that the Build Array function can have any number of inputs. Also, the 2D array could be created by nested loops.

second plot. The trick is to use some math to force those two data points to come out at the left and right edges of the plot.

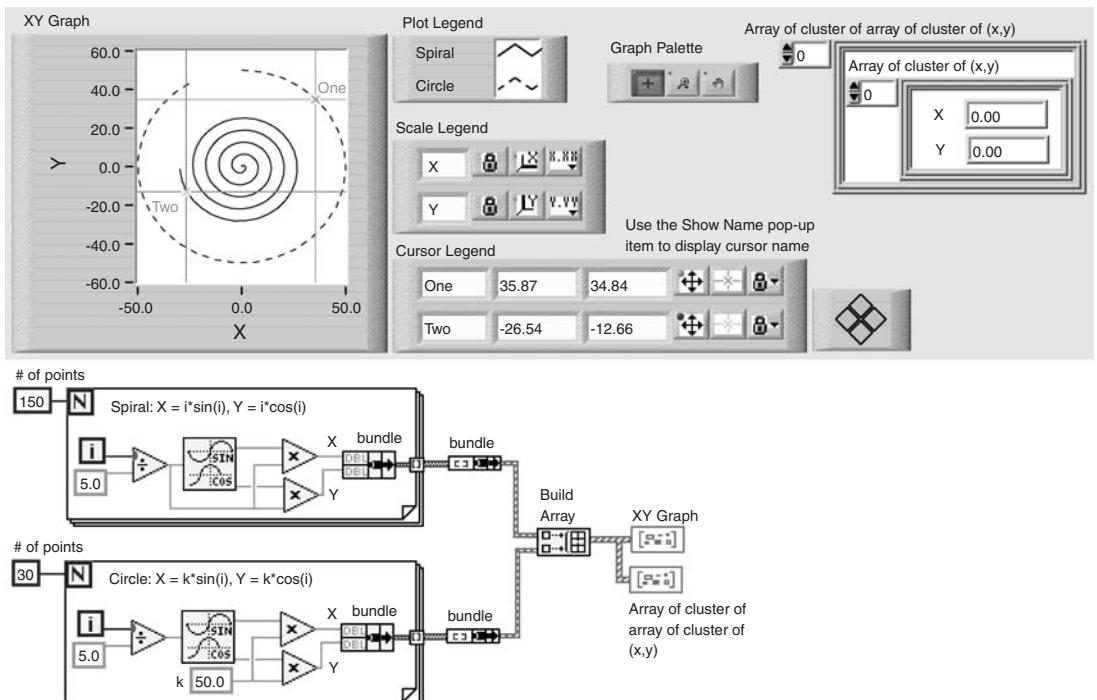
So far, we've been looking at graphs with simple, linear  $x$  axes which are nice for most time-based applications, but not acceptable for **parametric plots** where one variable is plotted as a function of another. For that, use the **XY Graph**. Figure 20.6 shows a way of building one of the data structures that defines a multiplot  $xy$  graph. A **plot** is defined



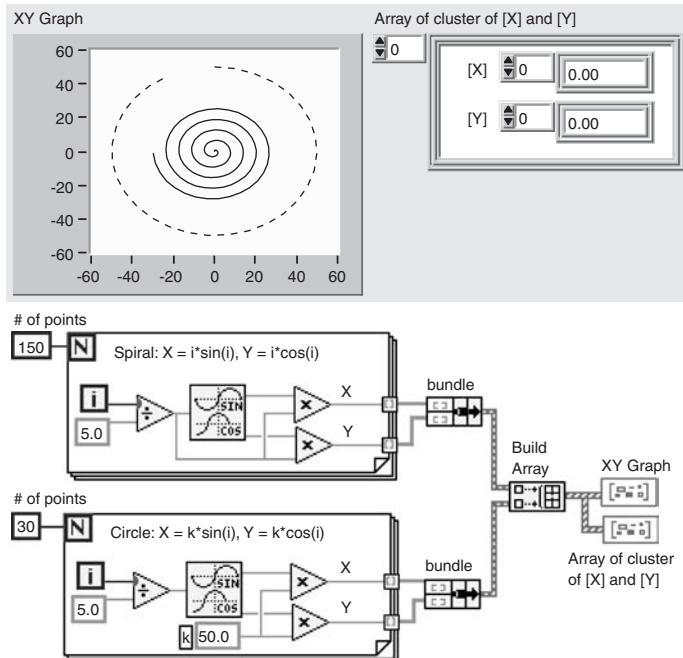
**Figure 20.4** Making multiple plots where each plot has a different number of samples. Build Cluster Array performs the same function as the combination in the dashed rectangle. Data could consist of waveforms instead of the 1D arrays used here.



**Figure 20.5** Yet another data structure for multiple plots accepted by the Waveform Graph. In this example, the sine wave is plotted with a line representing the mean value. This second plot has only two data points.



**Figure 20.6** One of the data structures for an XY Graph is shown here. It probably makes sense to use this structure when you gather  $(x,y)$  data-point pairs one at a time. This is the way it had to be done in LabVIEW 2.

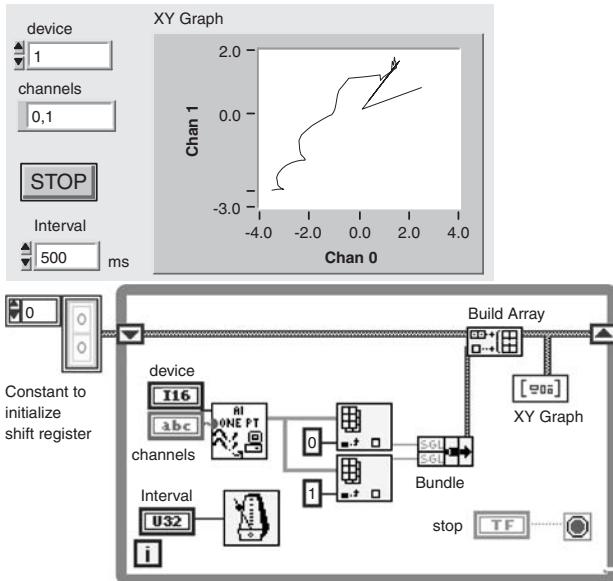


**Figure 20.7** Similar to the last XY Graph example (Figure 10.6), but using the other available data structure. This one is convenient when you have complete arrays of  $x$  and  $y$  data.

as an array of clusters that contains  $x$  and  $y$  data-point pairs. In this example, we turned on all the display options on the graph: the Plot Legend, Graph Legend, Graph Palette, and Cursor Legend. These are also available on the Waveform Graph.

Figure 20.7 shows how to use another available data structure for an  $xy$  graph. Here, a plot is defined as a cluster of  $x$  and  $y$  data arrays. Why would you use this format in place of the previous one? That depends on how your data is acquired or created. If you collect single data-point pairs, for instance, with a low-speed data recorder, then the previous format is probably best. For data that arrives in complete arrays for each plot, use the format in Figure 20.7.

A **real-time xy strip chart** can be constructed as shown in Figure 20.8. The trick is to add a new  $xy$  data pair to an ever-growing array that resides in a Shift Register. For each iteration of the While Loop, the acquired data is stored in the array, and all data is displayed in an  $xy$  graph. The biggest limitation is that performance decreases as time goes on because the amount of data to be displayed keeps growing. You also risk running out of memory. As an alternative, consider



**Figure 20.8** An XY Graph makes a handy real-time *xy* chart recorder. Keep appending data to an array stored in a shift register. *Caution:* You may run out of memory after extensive operation.

using the **Circular Buffer VI**, described in Chapter 18, “Process Control Applications.”

### Bivariate data

Beyond the ordinary cartesian data we are so familiar with, there is also **bivariate** data, which is described by a function of two variables, such as  $z = f(x,y)$ . Here are some examples of bivariate data that you might encounter:

- A photographic image is an example of bivariate data because the intensity ( $z$ ) changes as you move about the  $(x,y)$  area of the picture.
- Topographic data of a surface, such as the Earth, contains elevations ( $z$ ) that are a function of latitude and longitude ( $x,y$ ).
- A series of waveforms can be treated as bivariate data. For instance, the  $z$  axis can be amplitude, the  $x$  axis time, and  $y$  the sequence number. This is also called a **waterfall plot**.

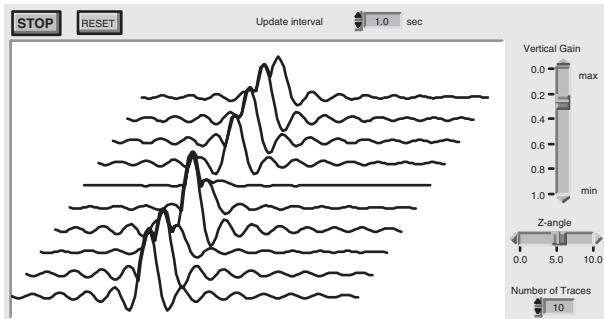
Bivariate data may be displayed in many formats. **Surface plots** have a three-dimensional (3D) look to them and are the most computationally intensive displays to generate. Surface plots may be solid

models (sometimes including shading) or wire frames with or without hidden lines removed. These plots can be very dramatic and give the reader insight into the general nature of the data, if properly displayed. The disadvantage of surface plots is that it is often difficult to use them quantitatively. Even with calibrated axes and the application of color, it's still hard to read out the value of a particular point as you would do on a simple  $xy$  graph. We'll look at a practical way of obtaining three-dimensional plots in LabVIEW in the next section.

Adding color to a surface plot effectively adds another dimension to the display, although it can be more confusing than helpful unless the data is well behaved. An example of effective use of color would be to display the topography of a mountain range and color the surface according to the temperature at each location. In general, temperature decreases with elevation, so a smooth progression of colors (or grays) will result. On the other hand, if you used color to encode wind speed at each location, you would probably end up with every pixel being a different color because of the rather uncorrelated and chaotic nature of wind-speed variations over large areas. Don't be afraid to try coloring your plots—just be sure that it adds *information*, not just glitz.

Bivariate data can also be displayed as an image. The **Intensity Graph** is a LabVIEW indicator that displays the value at each  $(x,y)$  location as an appropriate color. You choose the mapping of values to colors through a Color Array control, which is part of the Intensity Graph. This type of display is very effective for data such as temperatures measured over an area and, of course, for actual images. Use of the Intensity Graph is covered later in this chapter.

Yet another way to display bivariate data is to use a **contour plot**. A contour plot consists of a series of isometric lines that represent constant values in the data set. Familiar examples are topographical maps, maps of the ocean's depth, and isobars on a weather chart showing constant barometric pressure. LabVIEW offers limited contour plotting capability as part of the 3D Graph tools that we'll cover later. Many other data analysis and graphing packages already offer contour plotting, so you might plan on exporting your data. We might mention that contour plots are not as easy to generate as it might seem. The big problem is interpolation of the raw data. For instance, even though your measurements are acquired on a nice  $xy$  grid, how should the program interpolate between actual data points? Is it linear, logarithmic, or something else? Is it the same in all directions? Worse yet, how do you handle missing data points or measurement locations that are not on an even grid? Gary watched an analyst friend of his struggle for years on such a problem in atmospheric dispersion. Every change to the interpolation algorithm produced drastically different displays.



**Figure 20.9** A Waterfall Plot is a simple and efficient way to display waveforms that change with time.

You generally need some other display or analysis techniques to validate any contour plot.

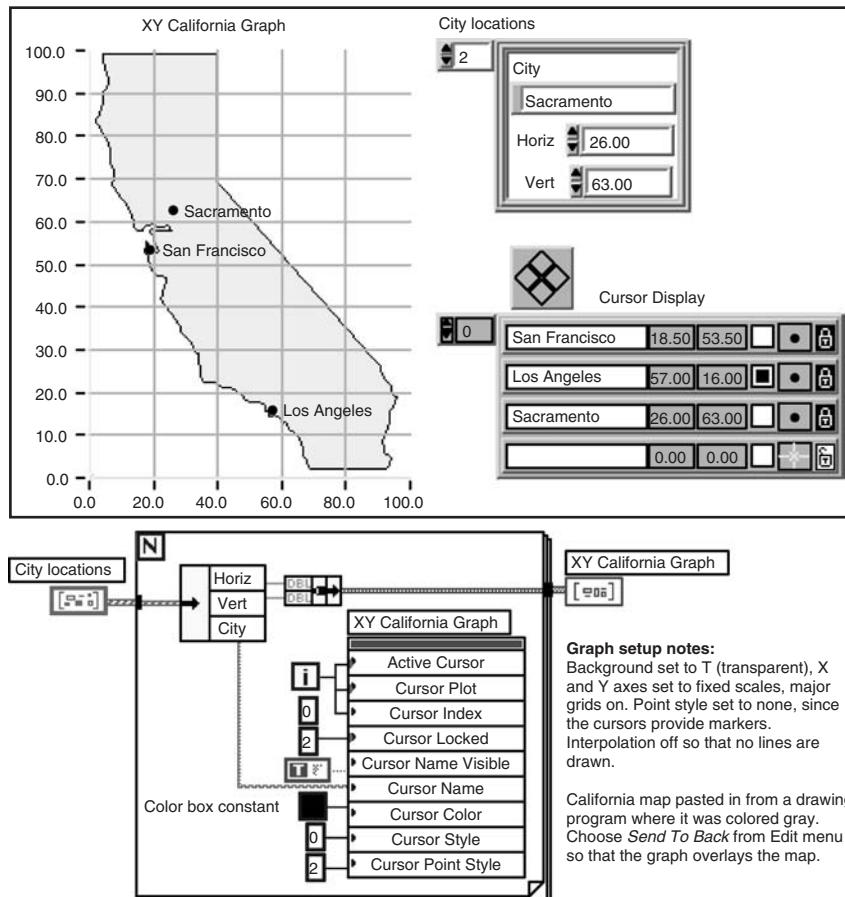
**Waterfall plots** are useful for observing a series of waveforms that change with time. For instance, you may want to observe the behavior of a power spectrum display while changing the system under observation. Figure 20.9 is a simple and efficient waterfall graphing VI that originally shipped with the LabVIEW 2 examples. It accepts a 1D array of data and does hidden-line removal.

### Multivariate data

Beyond bivariate data is the world of **multivariate data**. Statisticians often work in a veritable sea of data that is challenging to present because they can rarely separate just one or two variables for display. For instance, a simple demographic study may have several variables: *location*, *age*, *marital status*, and *alcohol usage*. All of the variables are important, yet it is very difficult to display them all at once in an effective manner.

The first thing you should try to do with multivariate data is analyze your data before attempting a formal graphical presentation. In the demographic study example, maybe the *location* variable turns out to be irrelevant; it can be discarded without loss of information (although you would certainly want to tell the reader that *location* was evaluated). Ordinary *xy* graphs can help you in the analysis process, perhaps by making quick plots of one or two variables at a time to find the important relationships.

Some interesting multivariate display techniques have been devised (Wang 1978). One method that you see all the time is a **map**. Maps can be colored, shaded, distorted, and annotated to show the distribution of almost anything over an area. Can you draw a map in LabVIEW? If you have a map digitized as *xy* coordinates, then the map is just an *xy* plot.



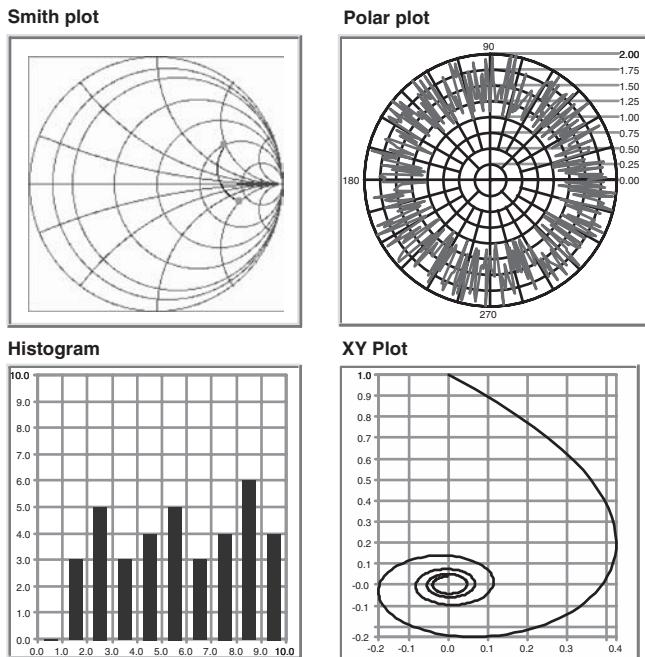
**Figure 20.10** Mapping in LabVIEW. We pasted in a picture (the California state map) from a drawing program, then overlaid it with an *xy* graph to display city locations. Cursor names annotate each city location.

Or, you can paste a predrawn map into your LabVIEW panel and overlay it with a graph, as we did in Figure 20.10. In this example, we created a cluster array containing the names and locations of three cities in California with coordinates that corresponded to the map. When the VI runs, the horizontal and vertical coordinates create a scatter plot. This might be useful for indicating some specific activity in a city (like an earthquake). Or, you could add cursors to read out locations, then write a program that searches through the array to find the nearest city. Note that a map need not represent the Earth; it could just as well represent the layout of a sensor array or the surface of a specimen. The hardest part of this example was aligning the graph with the map.

The map that we pasted into LabVIEW was already scaled to a reasonable size. We placed the *xy* graph on top of it and carefully sized the graph so that the axes were nicely aligned with the edges of the map. Through trial and error, we typed horizontal and vertical coordinates into the cluster, running the VI after each change to observe the actual city location.

City names are displayed on the graph with **cursor names**. On the diagram, you can see a Property Node for the graph with many elements—all cursor-related—displayed. You can programmatically set the cursor names, as well as setting the cursor positions, marker styles, colors, and so forth. This trick is very useful for displaying strings on graphs.

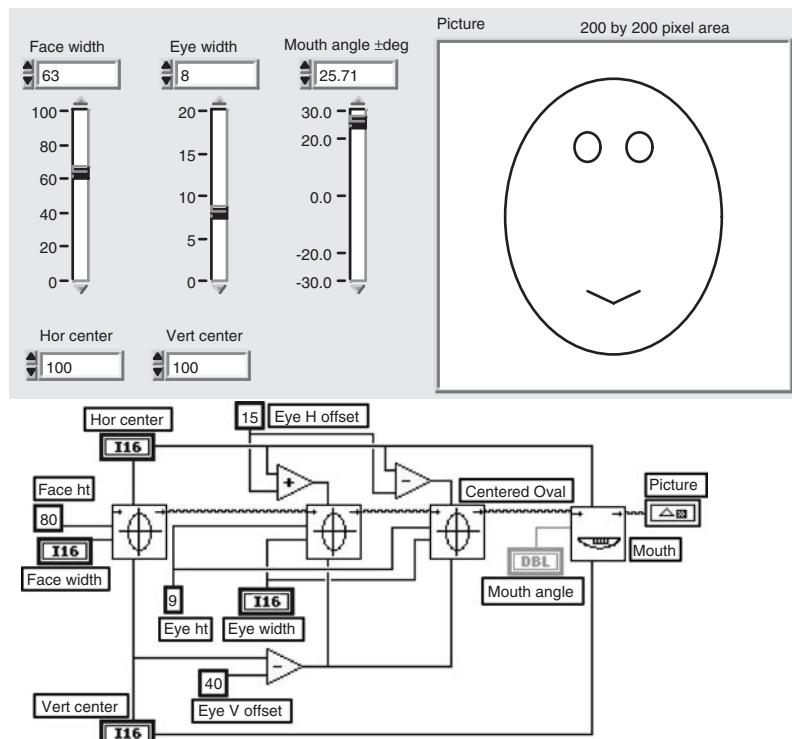
**Picture Controls.** The **Picture controls** add a new indicator, called a **Picture**, into which you can programmatically draw all types of geometric figures, bitmaps, and text. The toolkit includes many examples, most of them for the creation of specialized graphs with features that exceed those of the standard LabVIEW types. Figure 20.11 shows a few samples. The basic concept behind the Picture controls is that you start with an empty picture, then chain the various drawing VIs together,



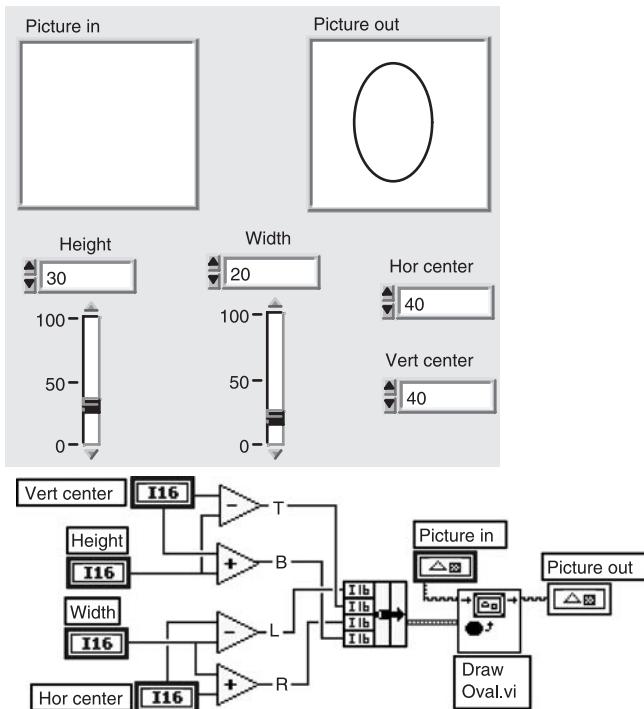
**Figure 20.11** Here are some of the specialized plots that you can draw with the Picture controls. These are included in the examples.

adding objects at each step. Each object requires specifications, such as position, size, and color. Let's take a look at an interesting example using this toolkit.

The most interesting multivariate data display technique we've heard of is the **Chernoff Face** (Wang 1978). The technique was invented by Herman Chernoff at Stanford University in 1971 for the purpose of simultaneously displaying up to 18 variables in the form of a simplified human face. His idea works because we are quite sensitive to subtle changes in facial expressions. Some applications where the Chernoff face has been successful are showing relative living quality in various metropolitan areas, displaying psychological profiles, and, my favorite, showing the state of Soviet foreign policy over the years. The trick is to map your variables into the proper elements of the face with appropriate sensitivity, and of course, to draw the face in a controllable manner. I spent some time with the VIs in the Picture controls, and managed to build a crude face (Figure 20.12) with three variables: face width, eye width, and mouth shape.



**Figure 20.12** A simplified Chernoff Face VI, with only three variables: face width, eye width, and mouth angle. This VI calls two subVIs that we wrote to draw some simple objects using the Picture Control Toolkit.



**Figure 20.13** This Draw Centered Oval VI does some simple calculations to locate an oval on a desired center point, then calls the Draw Oval Picture VI. Note the consistent use of Picture in–Picture out to maintain dataflow.

Because the Picture functions are all very low-level operations, you will generally have to write your own subVIs that perform a practical operation. In this example, we wrote one that draws a centered oval (for the face outline and the eyes) and another that draws a simple mouth (see Figure 20.13). All the Picture functions use *Picture in–Picture out* to support dataflow programming, so the diagram doesn't need any sequence structures. Each VI draws additional items into the incoming picture—much like string concatenation. Don't be surprised if your higher-level subVIs become very complicated. Drawing arbitrary objects is not simple. In fact, this exercise took us back to the bad old days when computer graphics (on a mainframe) were performed at this level, using Fortran, no less.

You could proceed with this train of development and implement a good portion of the actual Chernoff Face specification, or you could take a different tack. Using the flexibility of the Picture VIs, you could create some other objects, such as polygons, and vary their morphologies, positions, colors, or quantities in response to your multivariate data parameters. Be creative, and have some fun.

## 3D Graphs

Occasionally, data sets come in a form that is best displayed in three dimensions, typically as **wire-frame** or **surface** plots. When you encounter such data, the first thing you should do is try to figure out how to plot it in the conventional 2D manner. That will save you a good deal of effort and will generally result in a more readable graphical display. You should be aware that 3D graphing is much slower than 2D because of the extra computations required (especially for large data sets), so it's generally limited to non-real-time applications. Many data analysis applications have extremely flexible 3D graphing capability. We prefer to analyze our more complicated data with MATLAB, HiQ, IDL, or Igor.

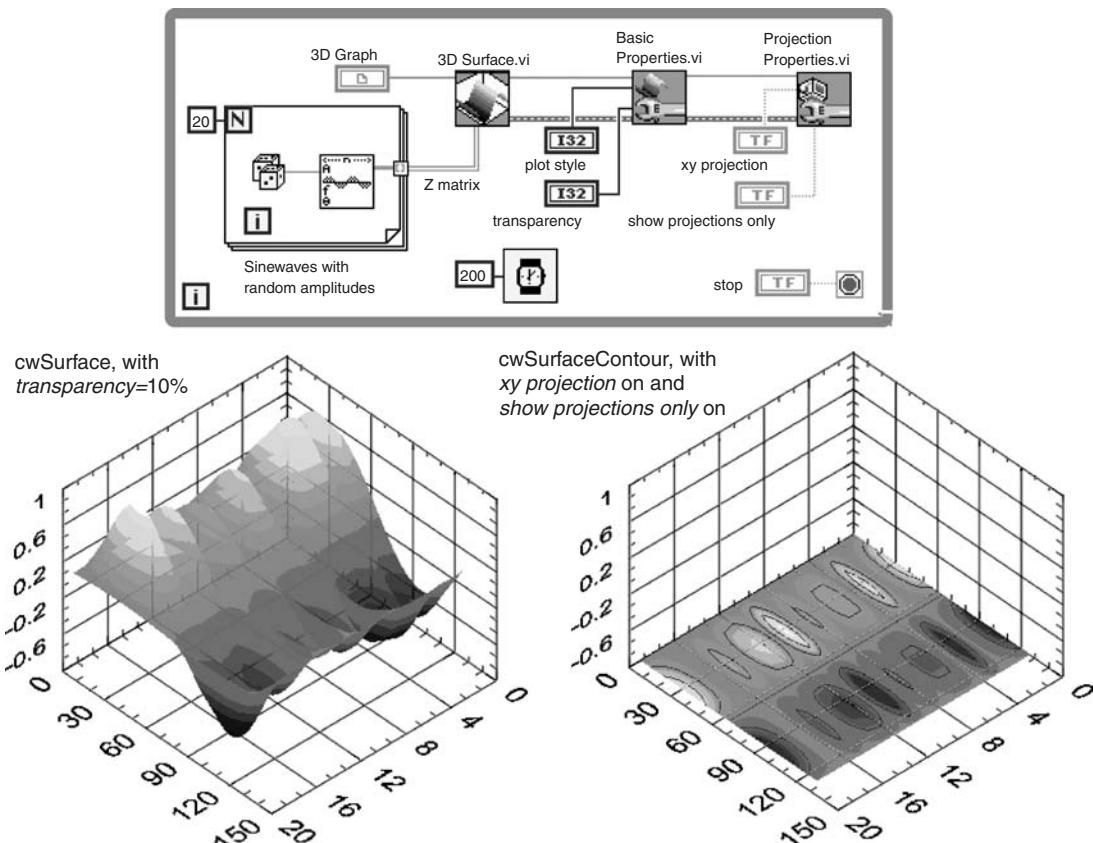
If you need to display 3D data in LabVIEW, you are limited to the Windows version, which includes **3D Graph** ActiveX objects that you drop on the front panel. A set of 3D Graph VIs manipulates that object through Invoke nodes and Property nodes. As with any 3D plotting problem, the real challenge is formatting your data in a manner acceptable to one of the available plot types.

Figure 20.14 shows the **3D Surface Graph** indicator displaying a waterfall plot and a contour plot. Data in this example consists of an array of sine waves with varying amplitude. In the real world, you might be acquiring waveforms from DAQ hardware or an oscilloscope. There are many possible plot styles, and a great deal of extra customization is possible through the use of the various 3D graph property VIs. A typical surface plot (plot style *cwSurface*) is shown with 10 percent transparency selected to allow the axes to peek through. You can also have the graph show projections to either of the three planes. With the plot style set to *cwSurfaceContour* and *xy* projection enabled, a shaded contour plot is produced.

Other available 3D display types are the **3D Parametric Graph** and **3D Curve Graph**. A parametric graph is most useful for geometric solid objects where 2D arrays (matrices) are supplied for *x*, *y*, and *z* data. If the data you are displaying looks more like a trajectory through space, then the 3D Curve Graph is appropriate. It accepts 1D arrays (vectors) for each axis.

There's a great deal of versatility in the 3D Graph, and you'll have to explore and experiment with the examples to see how your data can be massaged into a suitable form. Hopefully, the LabVIEW team will come up with a cross-platform solution to 3D graphing. Even when they do, the challenge of data formatting won't go away.

*Historical note:* The original 3D graphing package for LabVIEW was **SurfaceVIEW** from **Metric Systems**, a company started by Jeff Parker, who was one of the original LabVIEW team members.

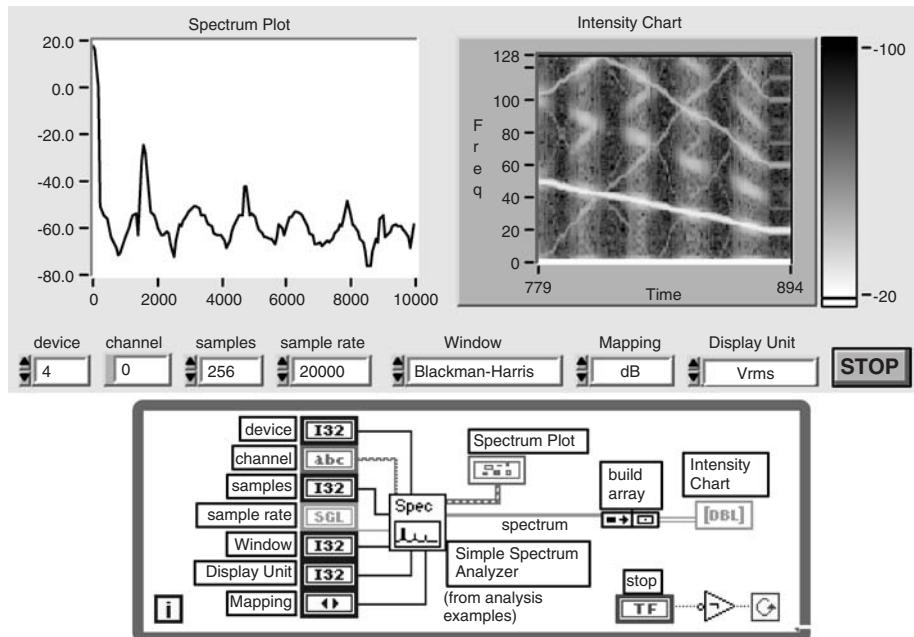


**Figure 20.14** Example of a 3D Surface Graph being used as a waterfall display. A rendered surface and a contour plot are shown.

Jeff informs us that the product is no longer available, but it certainly represents one of the first successful add-on packages that helped to make LabVIEW what it is today.

### Intensity Chart

Another interesting indicator, the **Intensity Chart**, adds a third dimension to the ordinary strip chart. It accepts a 2D array of numbers, where the first dimension represents points along the  $y$  axis and the second dimension represents points along the  $x$  axis, like a block-mode strip chart. Each value in the array is mapped to a color or intensity level. The biggest trick is setting the color or gray-scale range to accommodate your data. Fortunately, there is an option in the Intensity Chart pop-up for the  $z$ -scale called *AutoScale Z*. That gets you into the ballpark,



**Figure 20.15** The Intensity Chart in action displaying a power spectrum. White regions on the chart represent high amplitudes. Frequency of a triangle wave was swept from 4 to 2 kHz while this VI was running. The chart scrolls to the left in real time.

after which you can set the overall range and the breakpoints along the scale by editing the values on the color ramp. The colors can also be determined by an attribute node. This is discussed in the LabVIEW user's manual.

In Figure 20.15, the Intensity Chart gives you another way to look at a rather conventional display, a power spectrum. In this example, you can see how the power spectrum has changed over a period of time. The horizontal axis is time, the vertical axis is frequency (corresponding to the same frequency range as the Spectrum Plot), and the gray-scale value corresponds to the amplitude of the signal. We adjusted the gray-scale values to correspond to the interesting -20- to -100-dB range of the spectrum.

## Image Acquisition and Processing

Your LabVIEW system can be a powerful imaging tool if you add the right combination of hardware and software products. You will likely need hardware, in the form of a frame grabber and perhaps some video output devices, and software, in the form of image acquisition and processing VIs. The important thing to remember is that images are just

another kind of data, and once you're working in LabVIEW, that data is going to be easy to handle. In fact, it's no harder to acquire, analyze, and display images than it is to acquire, analyze, and display analog waveforms. Good news, indeed.

The basic data format for an image is a two-dimensional (2D) matrix, where the value of each element (a **pixel**) is the intensity at a physical location. **Image acquisition** involves the use of a frame grabber for live video signals or the use of mathematical transforms applied to other forms of data to convert it into a suitable 2D format. Standard image processing algorithms operate only on this normally formatted image data.

**Image processing** algorithms can be broadly grouped into five areas:

- **Image enhancement:** Processing an image so that the result is of greater value than the original image for some specific purpose. Examples: adjusting contrast and brightness, removing noise, and applying false color.
- **Image restoration:** A process in which you attempt to reconstruct or recover a corrupted image (which is obviously easier when you know what messed up the image in the first place). Examples: deblurring, removal of interference patterns, and correcting geometric distortion. The Hubble Space Telescope benefited from these techniques during its initial operation.
- **Image segmentation:** Dividing an image into relevant parts, objects, or particles. These algorithms allow a machine to extract important features from an image for further processing in order to recognize or otherwise measure those features.
- **Object representation and description:** Once an image has been segmented, the objects need to be described in some coherent and standardized fashion. Examples of these descriptive parameters are size, coordinates, shape factors, shape equivalents, and intensity distribution.
- **Image encoding:** Processing of images to reduce the amount of data needed to represent them, in other words, data compression and decompression algorithms. These methods are very important when images must be transmitted or stored.

Like signal processing, image processing is at once enticing and foreboding: There are so many nifty algorithms to choose from, but which ones are really needed for your particular application? When Gary first decided to tackle an imaging problem, he ran across the hall to visit a fellow EE and asked for a book. What he came back with was *Digital Image Processing* by Gonzales and Wintz (1987), which is the favorite

of many short courses in image processing. What you really need to do is study a textbook such as that one and get to know the world of image processing—the theory and applications—before you stumble blindly into a VI library of algorithms. Also, having example programs, and perhaps a mentor, are of great value to the beginner.

The most important imaging software for LabVIEW is **IMAQ Vision** from National Instruments. Years ago, **Graftek** developed **Concept V.i**, a highly modular package of CIN-based VIs that supports image acquisition and processing with interactive displays, file I/O, and a generally free exchange of data with the rest of LabVIEW. In 1996, National Instruments decided that imaging technology had reached the point where images could be acquired and manipulated just like data from a DAQ board. It reached an agreement with Graftek, purchased Concept V.i, and renamed it IMAQ Vision. You can now develop a flexible imaging application with LabVIEW and IMAQ as easily as you would write a data acquisition program with the DAQ library. Hence the name, **IMaging ACquisition**.

IMAQ has two family members offering different levels of image processing capability. The base package includes fundamental functions for image acquisition and file management, display and sizing utilities, histogram computation, and image-to-array conversions. The advanced package includes image processing and analysis functions such as lookup table transformations and spatial filters to allow for contrast and brightness adjustment, extraction of continuous and directional edges, noise removal, detail attenuation, texture enhancement, and so forth. It also includes frequency processing operations, arithmetic and logic operators, and object manipulations such as thresholding, erosion, dilation, opening, closing, labeling, hole filling, particle detection, and so forth.

Graftek will continue to support IMAQ Vision (as well as older versions of Concept V.i). Its specialty will be drivers for third-party frame grabbers, advanced and specialized algorithms, and custom application development. If you need special assistance with a custom imaging application, contact Graftek.

### System requirements for imaging

Basic graphing and simple image processing tasks are handled easily by any of the common LabVIEW-compatible computers, even the older, lower-performance Macs and PCs. But serious, real-time display and analysis call for more CPU power, more memory, and more disk space. Audio and video I/O adds requirements for specialized interface hardware that may not be included with your average computer, though the newer multimedia-ready machines sometimes have

decent configurations. Let's see what kinds of hardware you might consider above and beyond your generic data acquisition system.

**Computer configuration.** If it's images you're dealing with, then it's time to get serious about hardware. Image processing and display are positively the most CPU- and memory-intensive tasks you can run on your computer. Images contain enormous quantities of data. Many algorithms perform millions of calculations, even for apparently simple operations. And storing high-resolution images requires a disk drive with endless capacity. You can never afford the ideal computer that's really fast enough, or has enough memory, or big enough disks. But practical cost-performance trade-offs can lead you to a reasonable choice of hardware.

The fastest general-purpose computers available today come *close* to handling full-speed video, but be really cautious of manufacturers' claims. If you run custom-written applications with direct access to the I/O hardware, real-time video is possible on these faster machines. But if you expect LabVIEW to do lots of other things at the same time, such as running an experiment, there will be performance problems. As with high-speed data acquisition, buy and use specialized hardware to off-load these burdensome processing tasks, and you will be much happier with the performance of your system.

**Video I/O devices.** Unless your computer already has built-in video support (and LabVIEW support for *that*), you will need to install a **frame grabber** in order to acquire images from cameras and other video sources. A *frame grabber* is a high-speed ADC with memory that is synchronized to an incoming video signal. Most frame grabbers have 8-bit resolution, which is adequate for most sources. It turns out that cameras aren't always the quietest signal sources, so any extra bits of ADC resolution may not be worth the extra cost. Preceding the ADC, you will generally find an amplifier with programmable gain (*contrast*) and offset (*brightness*). These adjustments help match the available dynamic range to that of the signal. For the North American NTSC standard (RS-170) video signal, 640 by 480 pixels of resolution is common. For the European PAL standard, up to 768 by 512 pixels may be available, if you buy the right grabber. Keep in mind that the resolution of the acquisition system (the frame grabber) may exceed that of the video source, especially if the source is *videotape*. Both color and monochrome frame grabbers are available. Color images are usually stored in separate red-green-blue (RGB) image planes, thus tripling the amount of memory required.

Frame grabbers have been around for many years, previously as large, expensive, outboard video digitizers, and more recently in the

form of plug-in boards for desktop computers. With the birth of multimedia, frame grabbers have become cheaper and even more capable and are sometimes built into off-the-shelf computer bundles. The trick is to choose a board that LabVIEW can control through the use of suitable driver software or DLLs. Alternatively, you can use a separate software package to acquire images, write the data to a file, and then import the data into LabVIEW, but that defeats the purpose of building a real-time system.

In general, there are three classes of frame grabbers. First, there are the low-cost multimedia types, sometimes implemented as an external camera with a digitizer, such as the Connectix Quickcam. Also in this class are cheap plug-in boards and the built-in video capability in some Macintosh models. The primary limitations on these low-cost systems are that they include no hardware triggering, so you can't synchronize image acquisition with an experiment, and they often contain an autogain feature. Like automatic gain control in a radio, this feature attempts to normalize the contrast, whether you like it or not. This can be a problem for quantitative applications, so beware!

The second class includes midrange plug-in boards, such as the BitFlow Video Raptor, the IMAQ PCI-1408, and Scion LG-3. These boards have triggering, hardware gain and offset control, and various synchronization options. They offer good value for scientific applications.

High-performance boards make up the third class. In addition to the features of the midrange boards, you can install large frame buffers on these boards to capture bursts of full-speed images for several seconds, regardless of your system's main memory availability or performance. Special video sources are also accommodated, such as line scan cameras, variable-rate scanning cameras, and digital cameras. Some DSP functionality is also available, including built-in frame averaging, differencing, and other mathematical operations, in addition to full programmability if you're a C or assembly language hacker. Examples are the BitFlow Data Raptor, Imaging Technology IC-PCI, and the Scion AG5.

Now that you've got an image, how are you going to show it to someone? For live display you can use your computer's monitor from within LabVIEW or another application; or you might be able to use an external monitor. Hard-copy or desktop-published documents are always in demand.

One catch in the business of video output from your computer is the fact that high-resolution computer displays use a signal format (RGB component video) that is much different than the regular baseband NTSC or PAL standard format required by VCRs. There are several solutions (again, multimedia demands easy and cheap solutions, so expect to see some better ways of doing this). First, you can buy special converter boxes that accept various RGB video signals and convert

them to NTSC. An example is **TelevEyes** from Digital Vision Corporation, which converts the signals from many 640-by-480 pixel resolution RGB color displays to NTSC or S-video.

Another solution is to use a frame grabber board with video output capability (most boards have this feature). You write processed images to the output buffer, and the data passes through a DAC to create a standard video signal. You can also use computer display boards that generate NTSC outputs.

## Using IMAQ Vision

IMAQ Vision is very well designed, and the included example programs make it pretty clear how you put together an image acquisition and processing application. Also, the manual is quite thorough. You can also buy **IMAQ Vision Builder**, an IMAQ starter application developed by National Instruments. It includes a comprehensive set of image processing functions and makes it easy to try out various features of IMAQ. What we'll show you here are a few basic concepts and a real world application.

### IMAQ components

A complete IMAQ application really consists of two major parts. First is the image acquisition driver, called **NI-IMAQ**, that talks to the frame grabber. This driver, including its image acquisition boards, is free from National Instruments. If you want to use a non-NI board, you can often obtain driver VIs that are compatible with NI-IMAQ from Graftek. The NI-IMAQ driver is quite similar to the NI-DAQ driver, including the use of refnums and error I/O as common threads. You call the **IMAQ Init** VI to establish a new session with a channel on your board and then with various acquisition VIs that are organized in the familiar easy and advanced levels. If you've used DAQ, this is a piece of cake.

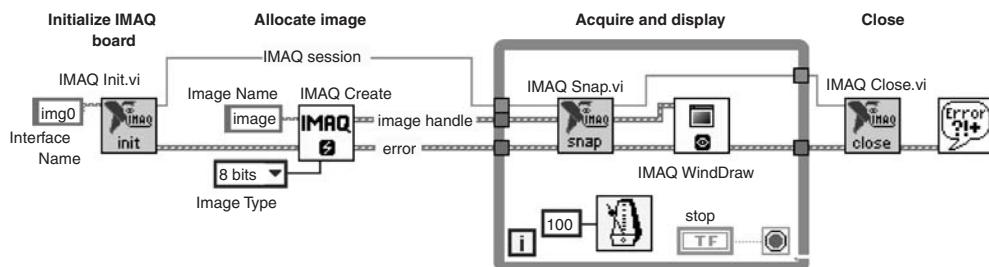
The other part of IMAQ is the **Vision** package, which is a very comprehensive image processing toolkit that costs about as much as LabVIEW itself. Without Vision, you're limited to the basic acquisition operations and some file I/O. With it, you're limited only by your knowledge of image processing. First-time users find that a bit daunting, because there are literally hundreds of functions in Vision, many of which have optional parameters that seem to require a Ph.D. to comprehend (what's a Waddel Disk Diameter, anyway?). Your best resources are the example VIs, the IMAQ manuals, and image processing textbooks. Even so, we've been pleasantly surprised at how easy it is to grab an image and extract key features. Let's look further into how you use IMAQ.

**Image handles.** IMAQ carries information about an image (name, size, resolution, data, and so forth) in a data structure that is accessible by each function in the IMAQ library but hidden from LabVIEW. To indicate which image you are working with, you use a cluster called an **image handle** which is a kind of refnum. Image handles are normally created by **IMAQ Create** and marked for destruction by **IMAQ Dispose**. You must never fiddle with the value of an image handle, or the program will cease to function (or *worse*). A new image handle must be created for every independent image. Like all data in LabVIEW, images (and their handles) are no longer accessible once the calling VI finishes execution.

One advantage to this image handle scheme is that you have explicit control of image creation and destruction, unlike the situation with LabVIEW's data types. If you create an image handle, load an image, then submit the handle to 10 different image processing functions from the IMAQ library, rest assured that there will be no duplication of the data array (image). In contrast, a LabVIEW array, if passed in parallel fashion to several different DSP subVIs, may require many duplicates. You might be able to chain some functions in series, if you're careful, thus eliminating much duplication.

**A simple acquisition and display application.** Figure 20.16 shows a basic IMAQ application that acquires and displays single frames with software timing. You begin by calling **IMAQ Init** with the desired *interface name*, which implies a board and channel number as defined in Measurement Explorer. That allocates resources in the NI-IMAQ driver and returns an IMAQ Session identifier that you pass to any other acquisition VIs. Next, you need to allocate memory for an image with the **IMAQ Create** VI, which returns an image handle.

At this point, you can begin an arbitrarily complex sequence of acquisition and processing steps. For instance, you might want to do a continuous, double-buffered acquisition based on the high-level **IMAQ**



**Figure 20.16** A simple IMAQ application that repeatedly grabs an image from a plug-in frame grabber and displays it.

**Grab Acquire VI** or its lower-level components. In our case, we'll use the simpler **IMAQ Snap VI** to acquire frames one at a time. With it placed in a While Loop, we get a frame every time the millisecond timer fires, or as fast as the frame grabber can go.

To display an image, you have several choices. The fastest and easiest way is to use the built-in IMAQ image windows. And the simplest way to do that is to call the **IMAQ WindDraw VI**. A whole family of display VIs let you manipulate the window's size, position, and color palette. Display windows can even be interactive (more on that later). Each time you call IMAQ WindDraw, the display is updated.

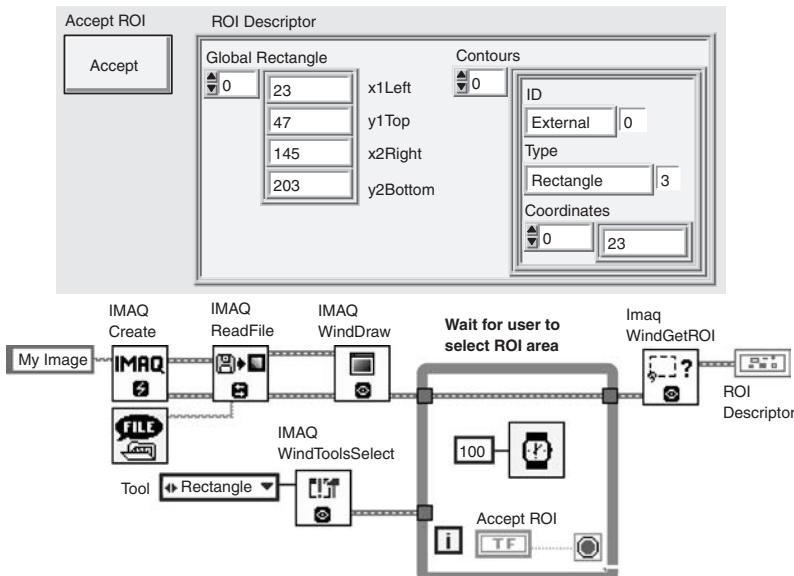
When the application is finished, you should call **IMAQ Close** to release the hardware resources. You can also call **IMAQ Dispose** to release the image memory, though this step actually is done for you behind the scenes. Display windows remain visible unless you call **IMAQ WindClose** or another display VI that explicitly makes the window disappear (the user can also manually close the window).

**Interactive windows.** Many applications require user interaction with the image. In IMAQ, users can draw arbitrary shapes, including text, or use a variety of tools to highlight a certain area of the image. For instance, you may want them to select a desired area for analysis, known as a **region of interest (ROI)**. IMAQ Vision includes a set of tools that permit the user to draw shapes and text on the image and then lets your program find out what they drew. There's even a tool palette that you can customize and show/hide programmatically.

Figure 20.17 shows a simple example of ROI selection. An image is read from a file and then displayed. The **IMAQ WindToolsSelect VI** chooses the tool type, in this case a rectangle. At any time, the user can click and drag in the image window. When you call the **IMAQ WindGetROI VI**, it returns the coordinates and type of ROI object that was drawn. Your real-world applications may become quite complex. If you need to manage a great deal of user interaction, consider writing a state machine that handles the various modes of operation and updates the image at the appropriate times.

**Image files.** IMAQ Vision has a set of VIs that make it easy to read and write files in standard image formats, including TIFF, JPEG, BMP, AIPD, and PNG. No data conversion steps are required. All you need to do is wire your image handle to **IMAQ Write File** or **IMAQ Read File**.

**An image processing example.** As we've said, image processing is a fairly complicated field of study. But there's at least one very familiar sequence of steps that we've seen so often that it qualifies as a

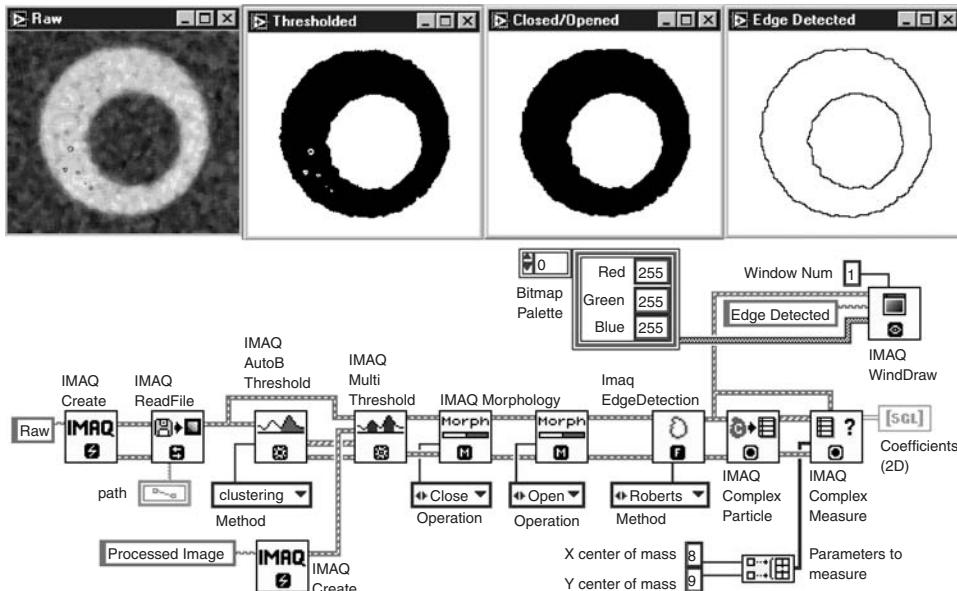


**Figure 20.17** The simplest example of reading the description of a user-selected region of interest.

canonical image processing example. What we're going to do is locate one or more important **particles** in the image. A particle is generally defined as any closed path. One way to do this is to use the sophisticated **pattern matching** VIs in IMAQ. In that case, you define a *template* by selecting a typical object in an image, and then the matching routine attempts to locate similar objects, regardless of their position or rotation. It's very powerful, and it is certainly worth considering. But for now we're going to concentrate on a more conventional stepwise approach that's very flexible and easy to optimize.

The first step is to turn an acquired gray-scale image into a simpler bitmap image. This step is known as **thresholding**, and there are many algorithms available to do it. Try using the **IMAQ AutoB-Threshold** VI, which features five different algorithms that you can test on your data. What's nice about auto-thresholding is that it does a good job of normalizing the image, thus reducing problems related to overall contrast and brightness. Figure 20.18 is a complete working example based on thresholding. It came from a project that Gary worked on involving the alignment of a telescope, where the objective was to locate the centroids of two concentric circles. Let's see how it works.

Starting with an acquired image (read from a file in this case), we apply IMAQ AutoBThreshold to locate the 50 percent threshold value.



**Figure 20.18** An image processing example that uses thresholding, morphology, and spatial filtering to locate concentric circles. It's deceptively simple.

Next, **IMAQ MultiThreshold** uses that threshold value to convert the grayscale image into a binary one. Instead of overwriting the original image, we chose to create a new one for the binary image that will be further processed. In the images in Figure 20.18, you can see how the original noisy pattern has been transformed into a fairly clear binary pattern. By default, the **IMAQ WindDraw** VI displays an image as grayscale. But if you want to display a binary image, you need to set the color palette as shown in the figure; otherwise, all you'll see is black.

Here's a trick for troubleshooting your image processing sequence. You can wire **IMAQ WindDraw** to any image on the diagram to see what has happened at that step. But something unexpected may appear: You always see the final result of the processing sequence no matter where you connect **IMAQ WindDraw**. Why is that? On a fast computer, a race condition may occur where the whole chain of VIs executes before **IMAQ WindDraw** actually copies image data to the window. Remember that the image is not duplicated when it's passed along through a chain of IMAQ VIs. If you single-step your program, you will see the expected image at the step where you've made the connection. A solution to this problem is to break the chain between VIs and wire the input to the following IMAQ VI to an empty image constant. Pop up on the VI's *Image Source* input and select *Create Constant*. That will keep the downstream chain happy without overwriting your other image.

Our thresholded image isn't perfect, as you can see, because it contains some holes in the dark area due to noise in the original image. There are many ways to filter out noise-related pixels and other small aberrations, but the techniques that work best in this case are **dilation**, **eroding**, **opening**, and **closing**:

- *Erosion* eliminates isolated background pixels (white specks on a black background).
- *Dilation* is the opposite of erosion (i.e., it fills in tiny holes).
- *Opening* is an erosion followed by a dilation.
- *Closing* is a dilation followed by an erosion.

You really have to see these *morphological transformations* in action to understand how they work, and you need to try various sequences of transformations on your image to see what works best. **IMAQ Morphology** is the VI that does all of these operations and more. In our case, we needed a *close* followed by an *open* to clean up the image.

Since our objective is to extract the concentric circles, the next step is edge detection, a kind of **spatial filtering** that locates contours in an image. The IMAQ EdgeDetection VI has several methods, and again you should experiment. The final image in the figure shows the resulting clean, concentric circles.

Our final step is measuring the locations of the circles. **IMAQ ComplexParticle** locates all the particles in a binary image and returns a set of statistics, called a *complex report*, for each one. Included in the output cluster array are such items as area, perimeter, a global rectangle, and statistical summations that are useful in other calculations. To make sense of this raw data, the complex report is passed to **IMAQ ComplexMeasure**. The image processing community has devised a large number of standardized ways to quantify various attributes of a particle, and this VI apparently does them *all*. You supply an array of parameters (attributes) that you wish to measure, and the VI returns a 2D array containing a value for each parameter on each particle. In this case, we asked for the *X* and *Y* centers of mass, or centroid, so the VI returns a 2 by 2 array. All you have to do is look at the values in the array to determine where the circles are in relation to each other.

As you can see, this apparently simple task of locating a couple of circles took quite a few steps. Also, you can imagine that no image processing routine is 100 percent bulletproof. Given a noisy or distorted image, it may fail. Do plenty of testing with real images, perhaps using Vision Builder, to optimize your program. Also consider assigning alarm limits to some of those complex measurements to validate the results of your algorithm if it's used in real time.

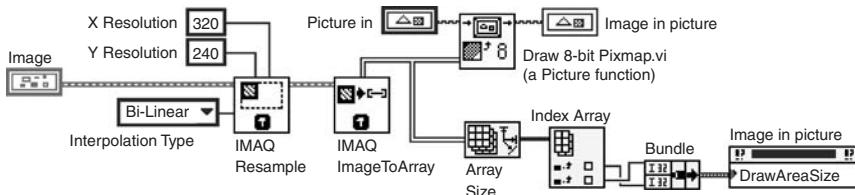


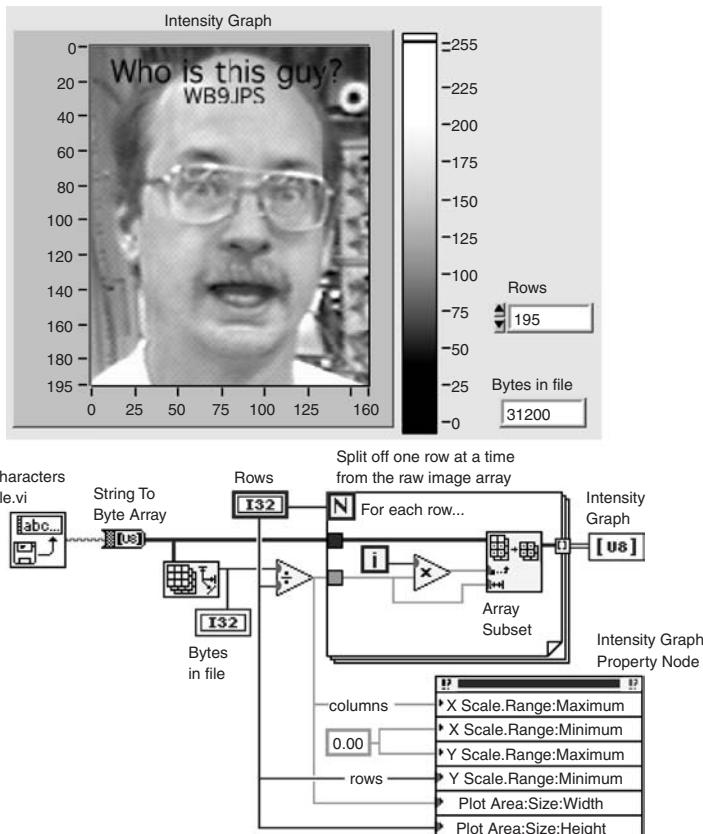
Figure 20.19 IMAQ images can be converted to LabVIEW Pictures for on-panel display.

**Other ways to display images.** You don't have to display your images in a floating IMAQ window, especially if you don't need any user interaction. A simple way to place the image on a LabVIEW panel is to use a **Picture** indicator. Figure 20.19 shows a practical example that includes **resampling** of the image to adjust its size without losing too much information. The **IMAQ Resample** VI lets you choose the X and Y resolution and one of four methods for interpolation. This step is not required if your IMAQ image is exactly the size that you want it to be in the Picture indicator. The next step is to transfer the image data from the special IMAQ format into a LabVIEW 2D array, which is done by the **IMAQ ImageToArray** VI. That's a useful function any time you need to manipulate image data in a manner not already included in IMAQ. Its complement is **IMAQ ArrayToImage**. Finally, you use one of the Picture functions such as the **Draw 8-bit Pixmap** VI to convert raw 2D data into a picture data type for display.

We added an extra feature to this example. The Picture indicator is automatically sized to fit the image. You get the array dimensions (corresponding to image width and height), bundle them up, and pass them to a Property Node for the Picture. The DrawSizeArea property is the one you need to set.

**Intensity Graph for built-in image display.** LabVIEW has another means by which you can display images right on the front panel: the **Intensity Graph**. The Intensity Graph accepts several data types, among them a 2D array corresponding to an image. We wrote a simple VI (Figure 20.20) to load a raw binary image from a file and format the data for display in an Intensity Graph. Because a raw binary file contains no header to describe the dimensions of the image, the user has to supply that information. The control called **Rows** supplies this key information. If you enter the wrong number, the image is hopelessly scrambled.

Looking at the diagram, the file is read as a string which is then converted to an array of U8 using the **String to Byte Array** conversion function. A For Loop breaks the long array of values into rows, which are then placed in a 2D array for display. An **Attribute Node** for the



**Figure 20.20** The Intensity Graph displaying an image from a binary file. The user has to know the size of the image—particularly the number of rows. The raw data string is converted to an array, which is then transformed into a 2D array suitable for display.

Intensity Graph sets the *y*- and *x*-axis scales to fit the actual number of rows and columns in the image. Note that the *y* axis has zero at the top, rather than its usual place at the bottom. That makes the image come out right-side up. You also have to select **Transpose Array** from the pop-up menu on the Intensity Graph to swap the *x*- and *y*-axis data.

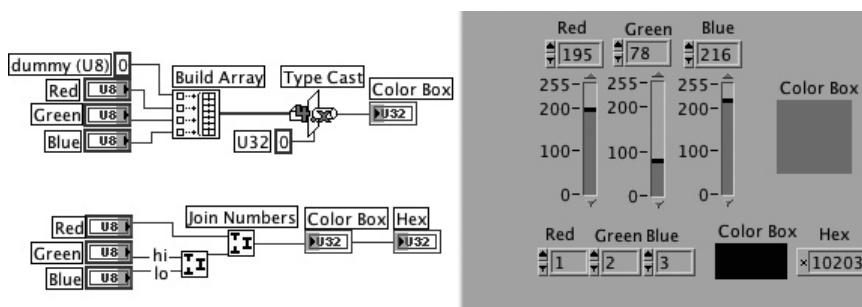
An important tweak that you can make is to match the number of displayed pixels to the data. Ideally, one array element maps to one pixel, or at least there is an integral ratio between the two. This reduces aliasing in the image. In this example, we made this adjustment by setting the Plot Area Size (*x* and *y* values) with the Property Node.

The Ramp control (part of the Intensity Graph) sets the data range to 0–200, corresponding to a pleasing gray scale ranging from black to white. You can adjust the gray scale or color gradations by editing the numeric markers on the Ramp. Pop up on the Ramp control and set

the marker spacing to *Arbitrary*. Pop up again to add or delete markers. You can then change the value of each marker and slide it up and down the ramp to create breakpoints in the (normally linear) mapping of intensity values to display brightness or color. For fun, you can pick various colors or put in palette breakpoints by adding elements to the Color Table, another item available through the Attribute Node. This is a means of generating a kind of color lookup table (CLUT).

Colors are stored as unsigned long integers (U32) with the following representation. The least-significant byte is the value of blue, the next byte is green, the next is red, and the most-significant byte is unused, but is normally set to zero. (In some applications, this extra byte is called the **alpha channel** and represents transparency.) For instance, 100 percent red (255 in the red byte) plus 50 percent green produces orange. This is also known as **32-bit color**. To select a color, you have several options. On the diagram, you can place a Color Box Constant (from the Additional Numeric Constants palette), then pop up on it to choose a color from those available in LabVIEW. A corresponding control on the panel is the Color Box constant from the Numerics palette. A related control, the Color Ramp, allows you to map an arbitrary number to an arbitrary color. All of these controls and constants use a visual, rather than numeric, representation of color. Remember that the actual color on your monitor is limited to the palette permitted by LabVIEW, which in turn is limited by your monitor setting.

For arbitrary colors, you must generate an appropriate U32 number. Figure 20.21 shows two of the possible ways to combine values for red, green, and blue into a composite RGB value. The data conversion functions (**Type Cast** and **Join Numbers**) come from the Advanced >> Data Manipulation palette. You might use this technique for combining separate RGB color image planes into a single, displayable color image. The complementary operation is also feasible, using the **Split Numbers** function to break a color into its RGB components.



**Figure 20.21** Colors can be generated programmatically, but it requires some data manipulation tricks.

## Sound I/O

Recording and reproducing sound can be useful for scientific analysis, operator notification, or as a novelty. The whole idea is to convert acoustic vibrations into an electrical signal (probably with a microphone), digitize that signal, and then reverse the process. In most cases, we use the human audible range of 20 Hz to 20 kHz to define sound, but the spectrum may be extended in both directions for applications such as sonar and ultrasonic work. Depending upon the critical specifications of your application, you may use your computer's built-in sound hardware, a plug-in board (DAQ or something more specialized), or external ADC or DAC hardware. As usual, a key element is having a LabVIEW driver or other means to exchange data with the hardware.

The simplest sound function that's included with LabVIEW is the **Beep VI**, found in the Sound function palette. On all platforms, it plays the system alert sound through the built-in sound hardware.

### DAQ for sound I/O

An ordinary DAQ board is quite capable of recording and reproducing sound up to its Nyquist sampling limit. The only technical shortcomings of such a board may be in the areas of harmonic and intermodulation distortion, the lack of built-in antialiasing filters, and perhaps a dynamic range limitation with 12-bit boards. For those reasons National Instruments developed its dynamic signal acquisition boards, featuring antialiased 16-bit inputs and outputs and very low distortion. In any case, all you need is the standard DAQ library to do the recording and playback in LabVIEW.

The maximum frequency that your system can process continuously is probably limited by CPU performance. The keyword is *process*: Do you want to do real-time filtering, FFTs, and display of signals with a 40-kHz bandwidth? You probably won't make it, at least with today's general-purpose CPUs and LabVIEW. Perhaps a DSP board can take care of the numerical processing; check out the DSP products from several vendors, but remember that extra programming—usually in C—is required. At lower frequencies, though, you can certainly build a usable real-time system. Be sure to run some benchmarks before committing to a real-time DAQ approach.

The good news about DAQ for sound applications is that it's easy to use. Review the analog I/O part of this book for details on single-shot and continuous waveform acquisition and generation. For widebandwidth signals, disk streaming is usually the answer; and again, the solutions are built into the DAQ and file I/O libraries.

A recurring issue with sound I/O is that of file formats and standards. On each platform, there are certain file formats that the

operating system can conveniently record and play back through standard hardware. Therefore, it's desirable to use those formats for your own, DAQ-based sound I/O. In the sections that follow, we'll look at some options for handling those formats.

## Sound I/O functions

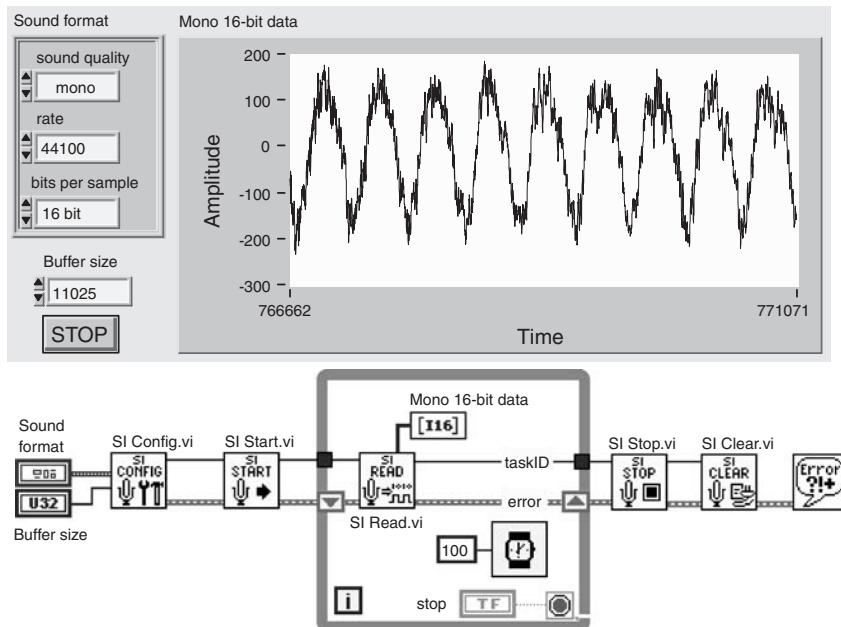
Once upon a time, there were no sound I/O functions built into LabVIEW, so we relied on a variety of third-party and donated VIs. Now we have a full array of VIs that provide multiplatform support for sound input, output, and various sound file formats. The sound I/O VIs support standard kinds of sound hardware that you'll find on any Macintosh or PC, such as the SoundBlaster. As long as your hardware can be accessed by the ordinary operating system interfaces, LabVIEW should be able to access it as well. Signal types include monaural and stereo in 8- or 16-bit formats using offset binary coding. Several sampling rates are available, including 11,025, 22,050, and 44,100 samples per second.

National Instruments made things easy for us once again by designing the sound I/O VIs much like the DAQ library. You configure a port, read or write data, and start/stop these buffered operations in the obvious way. Examples included with LabVIEW cover most of the features you'll be interested in using. Let's take a quick look at some input and output applications.

### Sound input

It's very easy to set up an input data streaming application, as shown in Figure 20.22. The **SI Config** VI allocates an input buffer and defines the type of data and sampling rate, much like its DAQ cousin, AI Config. Then you call **SI Start** to begin a continuous buffered acquisition. There's typically no triggering hardware available on consumer-grade sound boards, so the SI library does not support triggering. (Perhaps National Instruments could add software triggering like the NI-DAQ driver has.) Once the acquisition is started, you put the **SI Read** VI in a loop to read buffers as they become available. If you don't read often enough, you'll get a buffer overflow error. Adding a timer in the loop prevents the SI Read operation from taking up all the CPU time while it waits for the next available buffer. Data is returned as I16 or I8 arrays; stereo data is 2D. When you're done, call **SI Stop** and then **SI Clear** to free up memory.

If all you want to do is a single-shot acquisition of sound, there's a simpler VI in the sound palette, called **Snd Read Waveform**. It combines all the steps of Figure 20.22, but without the While Loop, and returns all possible data types (mono, stereo, 8- and 16-bit).



**Figure 20.22** A chain of sound input VIs performs a hardware-timed, buffered acquisition from your computer's built-in sound hardware.

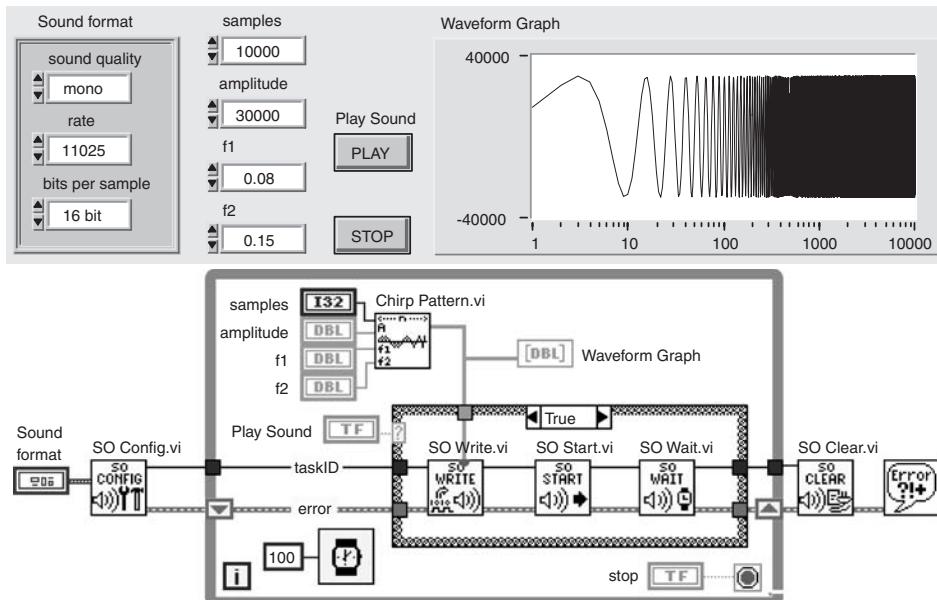
## Sound output

Sound output functions are similar to those for input, with the addition of a few extra features that help control the flow of data. In Figure 20.23, a waveform is computed and then written to the output buffer by the **SO Write** VI. In contrast with sound input behavior, this buffer size is variable and depends only on the amount of data that you pass to SO Write. Once the buffer is filled, you call **SO Start** to generate the signal. If the buffer is fairly long and you need to know when it's done, you can call the **SO Wait** VI, which returns only when the current sound generation is complete. It's also possible to call the **SO Pause** VI to temporarily stop generation; call SO Start again to restart it.

For single-shot waveform generation, you can use the **Snd Write Waveform** VI, which performs the configure, write, start, wait, and clear steps.

## Sound files

Included with the sound VIs are utilities to read and write Windows standard .wav files with all available data types and sample rates. They're called **Snd Read Wave File** and **Snd Write Wave File**, and



**Figure 20.23** Sound output is as easy as sound input. In this example we generate a cool sounding chirp each time the Play button is pressed.

they work on all platforms. The interface to these VIs meshes with the sound-in and sound-out VIs by including the sound format cluster and the four data array types.

For Macintosh users who need access to AIFF format files, Dave Ritter of BetterVIEW Consulting has created a CIN-based VI to play them. It's available at [www.bettervi.com](http://www.bettervi.com).

## Bibliography

- Cleveland, William S.: *The Elements of Graphing Data*, Wadsworth, Monterey, Calif., 1985.
- Gonzalez, Rafael C., and Paul Wintz: *Digital Image Processing*, Addison-Wesley, Reading, Mass., 1987.
- Tufte, Edward R.: *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Conn., 1983.
- Wang, Peter C. C.: *Graphical Representation of Multivariate Data*, Academic Press, New York, 1978.

*This page intentionally left blank*

# Index

“\” Codes Display, 110

## A

Abortable wait, 156  
Aborting, of For Loops, 67  
Absolute encoders, 523  
Absolute timing functions, 139, 145–146  
AC & DC Estimator, 544  
AC signals, 297  
Ac time domain, 280  
Accessibility, random, 177  
Acquire Semaphore VI, 164  
Action Instruments, 292  
Active filters, 292  
ActiveX objects, 126  
Actual scan rate, 499  
Actuators, 278–279  
Acyclic dataflow, 491  
A/D (analog-to-digital) system, 317  
Adaptive controllers, 446  
ADCs (*see* Analog-to-digital converters)  
Addressing, 254  
Adobe Illustrator, 237  
Aerotech Unidex, 523  
AI Clear, 532  
AI Config, 531  
AI CONVERT, 385  
AI Read, 531, 549  
AI Start, 531  
AI STARTSCAN, 385, 499  
Air Force Research Lab  
    *MightySat II.1*, 34  
Alarm acknowledge, 508  
Alarm annunciator panels, 512  
Alarm Handler VI, 509–511  
Alarm handlers, 508–511  
Alarm summary displays, 465  
Alarms, 506–512  
    alarm handler for, 508–511  
    for operator notification, 511–512  
Algorithms, 51  
Aliasing, 306  
Allen-Bradley:  
    Data Highway protocol, 457, 462  
    PLCs, 244  
Alliance Program, 244  
Alpha channels, 585  
Ames Research Center’s  
    HFFAF, 546  
Amperes (Current) controls, 78  
Amplifier(s), 291, 522  
    for instrumentation, 289  
    isolation, 291, 515  
    lock-in, 545  
    PGIA, 314  
Amplitude measurements, 280  
AMUX-64T, 525

AN084 (*Using Quadrature Encoders with E Series DAQ Boards*), 523  
AN114 (*Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability*), 145, 389–390  
AN154 (*LabVIEW Data Storage*), 132  
AN168 (*LabVIEW Performance and Memory Management*), 122, 548, 551  
Anafaze controllers, 462  
Analog design, 314  
Analog Devices 6B Series, 302, 303  
Analog filters, 308  
Analog IO Control Loop  
    (hw timed), 499  
Analog outputs, 299  
Analog signals, 305  
Analog transmitters, 447  
Analog voltage, 278  
Analog-to-digital converters  
    (ADCs), 309–314  
        characteristics of, 309  
        coding schemes for, 315–316  
        errors in, 311  
        multiplexing vs., 313  
        signals and, 275  
        use of, 305  
Analog-to-digital (A/D) system, 317  
AnaVIEW, 462  
Anderson, Monnie, 64  
Antialiasing, 292  
Antialiasing filter, 307  
Anti-reset window, 445  
AO Clear, 532  
AO Config, 531  
AO Start, 531  
AO UPDATE, 385  
AO Write, 531  
Aperture time, 313  
API (*see* Application program interface)  
Append True/False String  
    function, 107  
Apple Computer, 17  
Application Builder (LabVIEW), 25, 26  
Application design, 185–218  
    block diagrams for, 189–191  
    with DAQ Assistant, 197  
    debugging, 204–207, 214–218  
    design patterns for, 200–201  
    for error handling, 207–213  
    I/O hardware specification for, 191–192  
    with LabVIEW sketching, 202–203  
    for modularity, 198–200

problem definition, 186–191  
process for, 196–197  
pseudocoding, 203–204  
specifications definition, 187–189  
user interface prototyping for, 192–196  
user needs analysis for, 186–187  
with VI hierarchy, 201–202  
Application development  
    (for LabVIEW FPGA), 406–414  
        clocked execution, 411–413  
        compiling, 406–408  
        debugging, 408  
        parallelism, 413, 414  
        pipelining, 413–415  
        synchronization, 408–411  
Application events, 81  
Application Note 084 (*Using Quadrature Encoders with E Series DAQ Boards*), 523  
Application Note 114 (*Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability*), 145, 389–390  
Application Note 154 (*LabVIEW Data Storage*), 132  
Application Note 168 (*LabVIEW Performance and Memory Management*), 122, 548, 551  
Application program interface  
    (API), 28, 245  
Applications:  
    event-driven, 100–102  
    mega-applications, 22  
Applied Research Laboratory, 11  
Approach control gain  
    adjustment, 496  
Arbitrary waveform generator, 532  
Architecture, 87, 200  
Arnold, Tom, 34  
Array constant, 112, 113  
Array Size function, 115  
Array To Cluster function, 130, 135  
Array To Spreadsheet String, 111  
Arrays, 53–54, 114–122  
    clusters and, 124  
    coercion/conversion when  
        handling, 549  
    defined, 114  
    empty, 69  
FPGA (*see* Field-programmable gate arrays)  
initialization of, 117–119  
memory usage/performance  
    of, 119–122  
one-dimensional, 69  
shift registers and, 431  
two-dimensional, 112

Arrays of arrays, 124  
 ArrayToImage (IMAQ), 583  
 Artificial data dependency, 63  
 ASCII text files, 3, 167, 176  
*(See also Text files)*  
 ASCII text format, postrun analysis in, 328–329  
 AstroRT, 34  
 ATE (automated test equipment), 251  
 Attribute Node, 24, 583–584  
 Attributes, 126, 127  
 AutoBThreshold (IMAQ), 580  
 Auto-indexing, 66  
 Automated test equipment (ATE), 251  
 Automatic control, 444  
 Automatic error recovery, 261  
 Automatic parameter tuning, 496  
 Automatic save and recall, 367–370  
 Automation, 2–4  
 Autonomous VIs, 92–94  
 AutoScale Z, 571  
 Auto-setup, 272  
 Autotuning, PID with, 492  
 Average, 540  
 Averaging (signal sampling), 308–309  
 Avoidance, of sequence structures, 62

## B

Backups, 217  
 Balanced signals, 51, 290  
 Bancomm, 149  
 Bandwidth, signal, 343–344  
 Base clock, 407  
 Basic Averaged DCRMS VI, 125  
 BASIC programming, 9, 18  
 Batch controllers, 496  
 Batch processes, 446  
 Beep, 586  
 Benchmark products (LabVIEW), 34–36  
 Best practices (LabVIEW Embedded), 431–434  
 BetterVIEW Consulting, 589  
 Bias current, 296  
 Big-bang controllers, 444  
 Big-endian (byte ordering), 131  
 Binary files, 167, 169–170, 176–181  
*ASCII files vs., 176*  
*reading, 179–181*  
*writing, 178–179*  
 Binary formats, custom, 329–331  
 Binary-format byte stream files, 169–170 (*See also* Binary files)  
 Bipolar converters, 315  
 Bivariate data, on graphs, 563–565  
 Block diagrams, 12–13, 48, 189–191

Blocked (thread), 391  
 BMP (embedded image) files, 236  
 BNC-2120 accessory, 523  
 Board Support Package (BSP), 421  
 Boards:  
   DAQ, 148  
   interface, 149  
   plug-in, 253–254  
 Bogus data, 299  
 Boolean Array To Number function, 130  
 Boolean control, 478  
 Boolean logic functions, 486–487  
 Boolean strings, 262–263  
 Boolean table, 486  
 Boosting, priority, 389  
 Bottom-up structured design, 197–198  
 Boxcar averager, 518, 542  
 Brain board, 302  
 Brazing, 187  
 Breakpoints:  
   for debugging, 57–58  
   defined, 57  
   setting of, 57–58  
   Suspend When Called, 58  
 BridgeVIEW, 26  
 Brightness, 575  
 Brunzie, Ted, 214  
 BSP (Board Support Package), 421  
 Buffered I/O, circular, 374–375  
 Buffering, 119, 386–387  
 BUFG, 407  
 BUFGMUX, 407  
 Bug day approach, 19  
 Build Array, 114, 559  
 Build Cluster Array, 559  
 Build Path, 168  
 Build Timing Source Hierarchy VI, 141, 397, 398  
 Build VI, 425  
 Build Waveform function, 126  
 Built-in global variables, 73–75  
 Bundle By Name, 122, 468, 475  
 Bundle function, 122, 559  
 Byte stream files, binary-format (*see* Binary-format byte stream files)  
 Bytes at Serial Port, 57

## C

C codes:  
   for digital timing, 149  
   generator and generation of, 419, 420, 427–428  
   in LabVIEW Embedded, 422–424  
 C Runtime Library, 422–423  
 Cabling, 254  
 Calibration, of transient digitizers, 535–536  
 Call by reference nodes, 471  
 Call Libraries, 59, 434

CAMAC (Computer Automated Measurement and Control), 518  
 Cancellation, 171  
 Canvas (program), 236  
 Capacitive coupling, 286  
 Capacitor, 286  
 Cartesian data, on graphs, 558–563  
 CASE (*see* Computer-aided software engineering applications)  
 Case structures, 432  
 Caution (*see* Alarms)  
 C-callable routines, 149  
 CERN, 519  
 Certification exams, 218–229  
*(See also specific types, e.g.: Certified LabVIEW Developer)*  
 Certified Instrument Driver Developers, 244  
 Certified LabVIEW Architect (CLA), 217  
 Certified LabVIEW Associate Developer (CLAD), 218–221  
 Certified LabVIEW Developer (CLD), 218, 221–229  
*example questions for, 223–229*  
*grading tenets of, 222*  
 Change Log File Binding, 182  
 Change-of-state detection, 69, 507  
 Channel (top index), 120  
 Channel Wizard (DAQ), 26  
 Charge pump-out, 314  
 Chart History, 499  
 Chartjunk, 557  
 Charts:  
   control, 506  
   flowcharts, 13  
   intensity, 571–572  
   SFCs, 490  
   strip, 464–465  
 Chernoff face, 568, 569  
 CINs (*see* Code Interface Nodes)  
 Circular buffer, 500, 563  
 Circular buffered I/O, 374–375  
 CLA (*see* Certified LabVIEW Architect)  
 CLAD (*see* Certified LabVIEW Associate Developer)  
 Classes, of field devices, 448  
 CLD (*see* Certified LabVIEW Developer)  
 Clear Timing Source VI, 143  
 Clear to Send (CTS) line, 247  
 Clients, 90  
 Client-server, 90–94  
 Client-server model, 476  
 Clipping, 310  
 Clock(s):  
   base, 407  
   crystal, 148  
   time base, 316  
*(See also Timing)*

- Clocked execution, 411–413  
 Close (IMAQ), 579  
 Close File, 63, 64, 171  
 Close VIs, 272  
 Closing, 582  
 Clump, 41  
 Cluster Size, 130  
 Cluster To Array, 129, 130  
 Clusters, 50–52, 122–125  
     arrays and, 124  
     contents of, 262  
     Customize Control and, 51–53, 122  
     defined, 122  
     numeric types and, 115  
     saved as strict typedef, 51  
     strict typedef and, 122  
     in time-critical code, 434  
     typedefs and, 122  
 CMRR (common-mode rejection ratio), 290  
 Code boolean, 262  
 Code Interface Nodes (CINs), 58–59  
 Code width, 310  
 Codes:  
     insertion of other, into LabVIEW, 58–59  
     user-defined, 265  
     (See also specific types, e.g.: C code)  
 Coding schemes:  
     of Sequence structures, 62  
     for signal sampling, 315–316  
 Coercion:  
     arrays and, 549  
     conversion and, 128–129  
 Coercion dots, 115, 432, 550  
 Col values, 181  
 Color, 585  
 Columns, 116, 121  
 Command rate, 270  
 Common threads, 63–64  
 Common-mode rejection ration (CMRR), 290  
 Common-mode signal, 289  
 Communication(s):  
     with instrument drivers, 251–256  
 LabVIEW RT and, 398–399  
     of PLCs, 457–458  
     standards for, 245–249  
 CompactRio (cRIO), 404–405  
 CompactVision system, 405  
 Comparator, 317  
 Comparison functions, 507  
 Compiled help files, 232  
 Compiled rate, 408  
 Compile-time polymorphism, 50  
 Compiling, 406–408, 423–424  
 Complex reports, 582  
 ComplexMeasure (IMAQ), 582  
 ComplexParticle (IMAQ), 582  
 Compression, data, 503  
 Computer Automated Measurement and Control (CAMAC), 518  
 Computer languages, 8  
 Computer-aided software engineering (CASE) applications, 97, 98  
 Concatenate Strings, 106  
 Concept V.i program, 574  
 Conditional terminal, 65  
 Configuration cluster array, 467  
 Configuration management, 350–370  
     editors for, 352–362  
         interactive editors, 355–357  
         for menu-driven systems, 360–361  
         pop-up editors, 357–360  
         static editors, 353–355  
         status displays for, 360–361  
     recording in, 365–370  
         automatic save and recall, 367–370  
         manual save and recall, 367  
 Configuration VIs, 268–269  
 Conflicts, in namespace, 161  
 Connections, 284–304  
     amplifier use for, 291–294  
     differential, 289, 290  
     differential/single-ended, 288–291  
     grounding/shielding of, 284–291  
     of differential/single-ended connections, 288–291  
     electromagnetic fields and, 286–288  
     error sources in, 288  
     ground, 284–286  
     of input signals, 294–298  
         AC signals, 297  
         digital inputs, 297–298  
         floating analog inputs, 294–295  
         ground-referenced analog inputs, 294  
         thermocouples, 295–296  
     with networking, 303–304  
     of output signals, 298–299  
     required, 239  
     single-ended, 288  
 Connector pane picture, 239  
 Consistency, 44  
 Constants:  
     array, 112, 113  
     looping and, 431  
 Consumer events, 160, 210  
     dequeuing by, 210  
     design pattern with, 209  
 Cont Acq to File (scaled), 180  
 Context switching, 377, 393–394  
 Continuous control, 446, 492–499  
     control strategy for, 493–496  
     I/O values in, 496–497  
     PIC VIs and, 497–498  
     timing/performance limitations on, 498–499  
 Continuous data:  
     real time analysis with, 339–340  
     single-shot data vs., 338  
 Continuous time functions, 305  
 Continuous-value functions, 305  
 Contour plot, 564  
 Contrast, 575  
 Control(s):  
     automatic, 444  
     continuous, 446, 492–499  
         control strategy for, 493–497  
         I/O values in, 496–497  
         PIC VIs and, 497–498  
         timing/performance limitations on, 498–499  
     description documentation of, 232–233  
     feedback, 453  
     initializing, 482  
     Iteration, 540  
     manual, 444  
     process (see Process control applications)  
     sequential, 446, 486–492  
         boolean logic functions in, 486–487  
         GrafcetVIEW and, 490–492  
         initialization problems in, 489–490  
         state machine architecture for, 487–488  
 Control charts, 506  
 Control Editor (see Customize Control)  
 Control events, 82  
 Control flow, 38  
 Control limits, 505  
 Control Online Help function, 233  
 Control reference, 45  
 Control system architectures, 449–455  
     distributed control system, 449–451  
     with personal computers, 451–452  
     selection of, 453–455  
 Supervisory Control and Data Acquisition, 451  
 Controller(s):  
     adaptive, 446  
     batch, 496  
     big-bang, 444  
     desired, 467  
     Hysteresis On/Off Controller, 493  
     on/off, 444  
     PI, 445  
     PID, 444  
     predictive, 446, 496  
     proportional, 444  
     smart (see Smart controllers)  
 Controller faceplate displays, 465

Controller signals, 447  
 Controls (LabVIEW), 45, 46  
 Conversion(s), 127–135  
     arrays and, 549  
     coercion and, 128–129  
     of data from one type to another, 128  
     by Flatten to String, 132–133  
     type casting and, 134–135  
     type setting and, 129–132  
 Conversion speed, 310  
 Convert 7.x Data, 133  
 Converters:  
     bipolar, 315  
     time-to-digital, 518  
 Cooperative multitasking, 41–42  
 Copy, 169  
 CorelDraw, 236  
*Coriolis* spacecraft (Naval Research Lab), 34  
 Cost, 455  
 Count, 180, 183  
 Count Buffered Edges, 386–387  
 Count input, 173  
 Coupling, 286  
 Crash cart, 3  
 Crashing (computer), 454  
 Create... Property Node, 45  
 Create (IMAQ), 578  
 Create Control, 472  
 Create Notifier, 158  
 Create Rendezvous VI, 165  
 Create Semaphore VI, 163  
 Create SubVI, 198  
 Creep, feature, 25  
 cRIO (CompactRio), 404–405  
 Critical selections, 161  
 Crowbar (short-circuit), 516  
 Crystal clocks, 148  
 CTS (Clear to Send) line, 247  
 Current (amperes) controls, 78  
 Current measurements, 516–517  
 Current Messages, 510  
 Current shunts (current viewing resistors), 516  
 Current transformer, 516  
 Current viewing resistors (current shunts), 516  
 Currents, 296  
 Cursor names, 567  
 Curve Graph, 3D, 570  
 Custom binary formats, for postrun analysis, 329–331  
 Custom online help, 233–234, 242  
     for document distribution, 242  
     as documentation, 233–234, 242  
 Custom Probe, 55  
 Customize Control (Control Editor), 193  
     clusters and, 51–53, 122  
     sliders in, 466  
     use of, 193  
 CVI (LabWindows), 33  
 Cycles (closed loops), 491

## D

D/A (digital-to-analog) system, 317  
 DACs (*see* Digital-to-analog converters)  
 DAQ (*see* Data Acquisition Program)  
 DAQ Assistant, 197  
 DAQ boards, 148  
 DAQ cards, R-series, 403–404  
 DAQ Channel Wizard, 26  
 DAQ Library, 297  
 DAQ VIs, 125  
 DAQmx Create Timing Source VI, 141, 397  
 Data, 50–54, 103–135  
     arrays, 53–54, 114–122  
     bogus, 299  
     Cartesian, 558  
     checking of, 55, 56  
     clusters, 50–52, 122–125  
     compression of, 503  
     continuous, 338–340  
     conversion of, 128  
     conversions of, 127–135  
     Convert 7.x Data, 133  
     enumerated types, 133–134  
     multivariate, 565  
     numeric, 104–105  
     polymorphic, 68  
     for real time analysis, 341–342  
     single-shot, 338–339  
     strings, 105–114  
     typedefs, 52–53  
     volume of, 341–342  
     waveform, 558  
     waveforms, 125–127  
 Data Acquisition Program (DAQ), 323–375  
     boards for, 148  
     Channel Wizard in, 26  
     configuration management features of, 350–370  
     compilers for, 362–365  
     editors for, 352–362  
     recording in, 365–370  
     data analysis for, 325–343  
     postrun analysis for, 326–336  
         real time analysis, 337–343  
     low-speed example of, 370–372  
     medium-speed example of, 373–375  
     sampling in, 343–350  
         digital filtering for, 344–350  
         oversampling, 344  
         signal bandwidth and, 343–344  
     for sound I/O, 586–587  
     VIs for, 125  
     (*See also under* DAQ)  
 Data analysis, 325–343  
     postrun, 326–336  
         in ASCII text format, 328–329  
         configuration information for, 333–335  
     in custom binary formats, 329–331  
     database for, 335–336  
     in datalog file format, 326–327  
     Datalog File Handler VI for, 327–328  
     direct links in, 331–332  
     timestamps for, 332–333  
 real time, 337–343  
     with continuous data, 339–340  
     display performance for, 342–343  
     with single-shot data, 338–339  
     speed of, 340–341  
     volume of data for, 341–342  
 Data buffer, 119  
 Data dependency, 62–63  
 Data descriptor, 128  
 Data dictionary, 198  
 Data distribution (process control applications), 475–486  
     with display VIs, 479–482  
     with input scanners, 476–477  
     of output data, 477–479  
     real-time process control databases for, 484–485  
     using network connections, 482–484  
     validation and, 485–486  
 Data fields, variant, 126  
*Data Highway Plus* (Allen-Bradley), 457, 462  
 Data input, 179  
 Data Manipulation palette, 131  
 Data quantity:  
     memory optimization for, 547–553  
     in physics applications, 546–553  
     reduction of, 546–547  
 Data Range, 205  
 Data Scroll, 501  
 Data Server VI, 482  
 Data sheets, 300  
*Data Storage* (Application Note AN154), 132  
 Data storage/sampling, in transient digitizers, 534–535  
 Data strings, 132  
 Data structures, 51  
 Data terminal, 181  
 Data types:  
     in CLAD, 219  
     determination of, 182  
 Data VIs, 270–271  
 Data visualization (*see* Graphs)  
 Database(s):  
     for debugging, 421  
     for postrun analysis, 335–336  
     for process control, 484–485  
     real-time, 475, 476, 484  
 Database-driven software environment, 484

- Dataflow, 38  
 acyclic, 491  
 programming with, 38–39  
 structured, 13
- Dataflow diagrams, 13
- Datalog File Handler VI, 327–328
- Datalog files, 170, 181–183  
 postrun analysis in, 326–327  
 reading, 182–183  
 utilities for, 183  
 writing, 181–182
- Datalog refnum, 181
- Datalog-format file, 170 (*See also* Datalog files)
- Datalogging and Supervisory Control (DSC) module, 438, 458, 479
- DataSocket Read function, 483
- DataSocket Transport Protocol (dstp), 483
- DataSocket Write function, 483
- DataSockets, 303, 483
- Date time rec, 146
- Date/Time, Seconds To, 138, 146
- Date/Time In Seconds, 138
- dB (decibels), 291
- DBL (double-precision floating point) numeric type, 115
- DC Estimator, 544
- DC plasma diagnostics, 520
- DCM (Digital Clock Manager), 407
- DCRMS VI, 125
- DCS (distributed control system), 449–451
- Deadband, 279, 503–504
- Debug database, 421
- Debugging, 54–58  
 during application design, 204–207, 214–218  
 breakpoint setting for, 57–58  
 in CLAD, 220  
 data checking for, 55, 56  
 with execution highlighting, 57  
 generating info for, 427  
 of globals, 78–80  
 instrumented, 421  
 in LabVIEW FPGA, 408  
 of local variables, 80  
 on-chip, 421  
 with performance checking, 216–217  
 of property nodes, 80  
 with single-stepping, 55–57  
 with subVIs, 54–55  
 with Suspend When Called breakpoints, 58  
 with trace commands, 214–216  
 of VIs, 54
- Decibels (dB), 291
- Decimal String To Number, 108
- Decoupling, 42
- Default Directory constant, 169
- Default string, 180
- Delays:  
 interchannel, 313  
 lag, 444, 492  
 trigger, 537
- Delete, 169
- Delete From Array, 117
- Delimiter input, 111, 174
- Demodulation, 287
- Dequeued, 100, 210
- Derivative (rate), 445
- Derivative limiting, 497
- Derivative responses, 498
- Description pop-up menu, 232, 273
- Descriptions, user-defined, 265
- Design, 196  
 analog, 314  
 bottom-up structured, 197–198  
 in CLAD, 220  
 of high-voltage systems, 515–516  
 iterative, 196  
 of man-machine interfaces, 44  
 of SubVI, 220  
 top-down structured, 197  
*(See also Application design)*
- Design patterns, 87–102, 200–201  
 in CLAD, 220  
 client-server, 90–94  
 event-driven applications, 100–102  
 independent parallel loops, 89–90  
 initialize and then loop, 87–89  
 with producer/consumer events, 209  
 Queued Message Handler, 98–99  
 site machines, 94–98
- Design standards, 442–443
- Desired controller, 467
- Desktop PCs:  
 flat hierarchy in LabVIEW use of, 406  
 local variables in LabVIEW use of, 406  
 parallelism in LabVIEW use of, 406  
 for real time, 379, 381  
 subVIs in LabVIEW use of, 406
- Destination Code, 509
- Destroy Notifier VI, 159
- Destroy Rendezvous VI, 165
- Destroy Semaphore VI, 164
- Detailed diagram help, 239
- Determinism, 378
- Development (LabVIEW), 15–18
- Diagrams:  
 block, 12–13, 48, 189–191  
 dataflow, 13  
 documentation of, 234  
 help with, 239  
 instrument loop, 442  
 for piping instruments, 439–442  
 state, 13, 97
- Dialog boxes, 261
- Differential connections, 288–291
- Differential linearity, 312
- Digital Clock Manager (DCM), 407
- Digital filtering, 344–350  
 analog filters and, 308  
 FIR filters, 345–347  
 IIR filters, 347–348  
 median filters, 348  
 moving averages, 348–350  
 timing for, 350
- Digital Image Processing* (Gonzalez and Wint), 573–574
- Digital inputs, 297–298
- Digital multimeter, 250
- Digital storage oscilloscopes (DSOs):  
 flash converters used in, 311  
 history of, 272  
 for pulse and transient phenomena, 536–539  
 uses of, 536
- Digital timing, 149
- Digital Vision Corporation, 577
- Digital-to-analog converters (DACs), 314–315  
 characteristics of, 309  
 coding schemes for, 315–316  
 for pulse and waveform generation, 403
- Digital-to-analog (D/A) system, 317
- Digitizers, transient (*see Transient digitizers*)
- Dilation, 582
- Dimension size, 117, 119
- Direct memory access (DMA), 5, 320, 373–374
- Discrete-time functions, 305
- Discrete-value functions, 305
- Discriminator, 317, 537
- Dispatcher, 481
- Dispatcher loop, 470
- Display hierarchy (MMIs), 469–473
- Display VIs (as clients), 479–482
- Displays:  
 in IMAQ Vision, 583  
 performance of, 342–343  
 status, 360–361
- Dispose (IMAQ), 578, 579
- Distributed control system (DCS), 449–451
- Distributed I/O, 302
- Distribution:  
 of documentation, 241–242  
 with HTML, 242  
 with PDF, 242  
 using custom online help, 242
- Dithering, 318
- DLLs (*see Dynamic Link Libraries*)
- DMA (*see Direct memory access*)
- Documentation, 231–242  
 in CLAD, 220  
 CLD graded on, 222  
 connector pane picture in, 239  
 of control descriptions, 232–233  
 custom online help as, 233–234, 242  
 of diagrams, 234

Documentation (*cont.*)  
 distribution of, 241–242  
 formal, 238–241  
 of instrument drivers, 273–274  
 outline of, 238–239  
 printing, 235–238  
 programming examples in, 241  
 screen images, 236–237  
 of terminal descriptions, 240–241  
 VI descriptions and, 231–232,  
   239, 240  
 of VI history, 234–235  
 writing, 238–241  
 DOS systems, 12  
 Double-buffered DMA, 320  
 Double-precision floating point  
   (DBL) numeric type, 115  
 Dr. T (*see* Truchard, Jim)  
 Draw 8-bit Pixmap, 583  
 Drawing standards, 442–443  
 Drivers, 191 (*See also* Instrument  
   drivers)  
 DSC (*see* Datalogging and  
   Supervisory Control module)  
 DSOs (*see* Digital storage  
   oscilloscopes)  
 DSP board, 320  
 dstp (DataSocket Transport  
   Protocol), 483  
 Dynamic events, 85–87, 154, 212  
 Dynamic Link Libraries (DLLs),  
   29, 59, 149  
 Dynamic links, 45  
 Dynamic memory allocation, 431  
 Dynamic user-defined events, 82

## E

Ectron, 292  
 Edit Format String, 110  
 Editors, 352–362  
   interactive, 355–357  
   for menu-driven systems,  
   360–361  
   pop-up, 357–360  
   static, 353–355  
   status displays for, 360–361  
 EG&G Automotive Research, 301  
 Elapsed Time, 222  
 Electromagnetic fields, 286–288  
 Electronics WorkBench Multisim, 32  
 Electrostatic shield, 286  
 Elemental I/O, 429  
 Embedded Development Module  
   (LabVIEW), 419  
 Embedded image (BMP) files, 236  
 Embedded LabVIEW (*see*  
   LabVIEW Embedded)  
 Embedded Project Manager,  
   424–425  
 Embedded Target Creation  
   Utility, 426  
 EMF objects, 236

Empty Array (function), 118, 119  
 Empty arrays, 69, 118  
 Empty Path, 169  
 Empty String/Path, 269  
 Emulation mode, 408  
 Enable chain, 410  
 Enable Database Access, 182  
 Encapsulated PostScript (EPS)  
   files, 236  
 Encoders:  
   absolute, 523  
   incremental, 523  
   PWM, 404  
   quadrature, 523  
   reading of, 523  
 Encoding images, 573  
 End or identify (EOI) signal, 254  
 End-of-line (EOL), 206  
 Enhancement, image, 573  
 Enqueued, 91, 100, 210  
 Ensemble averager, 540  
 Enterprise Connectivity Toolset,  
   505–506  
 Enumerated constraints  
   (Enums), 96  
 Enumerated types (Enums),  
   133–134  
 Enums, 133–134  
 EOF function, 183  
 EOI (end or identify) signal, 254  
 EOL (end-of-line), 206  
 Epoch time, 145–147  
 EPS (encapsulated PostScript)  
   files, 236  
 Equivalent time sampling  
   (ETS), 543  
 Equivalent time-sampling, 306  
 Erosion, 582  
 Error(s):  
   in actuators, 279  
   in ADCs, 311  
   automatic recovery of, 261  
   gain, 311  
   in grounding/shielding  
     connections, 288  
   offset, 311  
 Error Code File Editor, 209  
 Error Event VI, 212  
 Error handler, 208  
 Error handling:  
   application design for, 207–213  
   in CLAD, 220  
 Error in, 262, 263  
 Error I/O, 64, 257  
   flow control drivers, 261–265  
   in VISA functions, 272  
 Error out, 262, 263  
 Error Query, 265  
 Error Queue, 210  
 E-Series DAQ boards, 523  
 Ethernet adapters, 304  
 Ethernet Device Server,  
   246–247  
 Ethernet devices, 245, 457  
 ETS (Equivalent time  
   sampling), 543  
 Event driven programming, 434  
 Event handling, functional globals  
   and, 212  
 Event output, 72  
 Event structure, 81  
 Event-driven applications,  
   100–102  
 Events, 81–87  
   controls for, 85  
   defining, 81, 151, 153  
   dynamic, 85–87  
   dynamic user-defined, 82  
   filter, 83–85  
   Notify, 153  
   notify, 81–83  
 Sub Panel Pane, 81–82  
   synchronization of, 153–155  
   user-defined, 212  
 Value Change, 86–87  
 VI, 81  
 Excel (Microsoft), epoch  
   time in, 147  
 Excitation, 293  
 Exclusion, trigger, 539  
 Executable code, 420, 423  
 Execution:  
   of timing, 143–145  
   of VIs, 54  
 Execution highlighting, 57  
 Execution overhead, 62  
 EXIT command, 67  
 Exp String To Number  
   function, 109  
 Expansion, 453  
 Exponent, 128  
 Exporting timing data, 146–147  
 Extended precision (EXT)  
   float, 549  
 External IOBs (input/output  
   blocks), 407  
 External timing, 498, 499  
 External triggers, 316–317  
 Extract Single Tone Information  
   VI, 148

## F

Faceplate displays, 465  
 False String function, 107  
 Familiarization, 249–250  
 Fan-out, trigger, 537  
 Faraday cage, 286  
 Fast Fourier Transform (FFT), 281  
 FASTBUS, 519  
 Feature creep, 25  
 Feedback, 444  
 Feedback control, 453  
 Feedforward, 495  
 Femtosecond laser cutting  
   system, 34  
 Fencepost, 503

- FFT (Fast Fourier Transform), 281  
 Field bus, 448  
 Field diagnostics, 520–527  
     ion beam intensity mapper  
         application, 524–527  
     motion control systems for,  
         520–523  
     reading encoders and, 523  
 Field mapping, 520  
 Field-Point, 303  
 Field-programmable gate arrays  
     (FPGA), 30, 406 (*See also*  
         LabVIEW FPGA)  
 FIFO (First In, First Out), 399  
 File Dialog function, 169, 173  
 File mark, 179  
 File refnum, 169  
 File transfer protocols (FTP), 483  
 File/Directory Info, 169  
 Files, 167–183  
     accessing, 167–169  
     ASCII text, 167  
     binary, 169–170, 176–181  
         reading, 179–181  
         writing, 178–179  
     datalog, 170, 181–183  
     help, 232  
     in IMAQ Vision, 579  
     as mailboxes, 484  
     object, 423  
     for sound I/O, 588–589  
     text, 169–176  
         formatting to, 175–176  
         reading, 172–174  
         writing, 170–172  
     types of, 169–170  
 Fill button, 473  
 Fill frame, 488  
 Filter events, 83–85, 153  
 Filtering:  
     in signal sampling, 307–309  
     spatial, 582  
 Filters, 291  
     active, 292  
     analog, 308  
     for antialiasing, 307  
     digital, 308, 344–350  
         FIR filters, 68, 345–347  
         IIR filters, 347–348  
         median filters, 348  
         moving averages, 348–350  
         timing for, 350  
     low-pass, 307  
     reconstruction, 315  
 Find command, 75, 80  
 Finite impulse response (FIR)  
     filters, 68, 345–347  
 First Call, 222  
 First In, First Out (FIFO), 399  
 First row output, 173  
 First-level triggers, 537  
 Flags, 152  
 Flash converters, 311  
 Flat hierarchy, 406  
 Flatten to String, 132–133  
 Floating analog inputs, 294–295  
 Floating point operations, 432  
 Floating windows, 24  
 Floating-point (real) value, 104  
 Flow control, 61–102  
     common threads for, 63–64  
     data dependency and, 62–63  
     design patterns for (*see* Design  
         patterns)  
     error I/O, 261–265  
     events and, 81–87  
         controls for, 85  
         dynamic, 85–87  
         filter, 83–85  
         notify, 81–83  
     globals and, 71–80  
         debugging of, 78–80  
         global variables, 71–75  
         local variables, 75–78, 80  
     with looping, 64–71  
         For Loops, 66, 67  
         pretest While Loops, 65  
         shift registers for, 67–71  
         stops for, 65, 66  
         While Loops, 65–66  
     with sequences, 61–62  
 Flowcharts, 13  
 Fluke-Phillips, 514  
 Flush File, 172  
 For Loops, 64, 66, 67  
 Force update, 478  
 Formal documentation,  
     238–241  
 Format Date/Time String, 146  
 Format Into File, 175  
 Format Into String, 107, 110, 134  
 Format menu, 132  
 Format String input, 109, 175  
 Format Value, 106  
 Formatting, 175–176  
 Foundation Fieldbus, 303, 448  
 Four P's, 437  
 FPGA (*see* Field-programmable gate  
     arrays; LabVIEW FPGA)  
 FPP (Front Panel Protocol), 380  
 Fract/Exp String To Number  
     function, 109  
 Frame grabbers, 575–576  
 Frequency domain, 281  
 Frequency foldback, 306  
 Frequency-selective voltmeter, 544  
 Front Panel Protocol (FPP), 380  
 Front-panel, 44–45, 474–475  
 FTP (file transfer protocols), 483  
 Functional globals, 65, 71, 212  
 Functional requirements and  
     specifications, 187  
 Functionality (CLD), 222  
 Functions (CLAD), 219  
 Future predictions, for  
     LabVIEW, 32  
 Fuzzy logic, 446, 496
- G**
- G language, 26  
 G Math Toolkit, 550  
*G Programming Reference Manual*  
     (LabVIEW 5), 390  
 G Wizards, 459  
 Gain errors, 311  
 Gain scheduling variable, 496  
 Gape Applied Sciences, 533  
 Gate input, 523  
 Gated integrator, 518  
 Gates, Bill, 454  
 General Error Handler VI, 209, 264  
 General Purpose Interface Bus  
     (GPIB), 9, 245, 247, 248  
     hardware/wiring issues of,  
         253–254  
     history of, 9  
     protocols/basic message passing  
         of, 254  
     voltage and, 278  
 Generate Occurrence, 155  
 GenerateCFunctionCalls, 428  
 GenerateDestinationFolder, 427  
 GenerateGuardCode, 427  
 GenerateIntegerOnly, 427  
 GenerateSerialOnly, 427  
 Generator, arbitrary waveform, 532  
 Get Date/Time In Seconds, 146  
 Get Date/Time String, 138, 146  
 Get Waveform Attributes, 127  
 Get Waveform Components  
     function, 125  
 Global Alarm Queue VI, 508–509  
 Global Positioning System (GPS),  
     148, 149  
 Global queue, 210, 508  
 Global References, 75  
 Global variables, 71–75  
     built-in, 73–75  
     local variables vs., 431  
     VI for, 71  
 Globals, 71–80  
     debugging of, 78–80  
     functional, 65, 71, 212  
     global variables, 71–75  
     local variables, 75–78, 80  
 GOTO command, 67  
 GPIB (*see* General Purpose  
     Interface Bus)  
 GPIB-ENET, 304  
 GPS (*see* Global Positioning  
     System)  
 GPS interface boards, 149  
 Grab Acquire (IMAQ), 578, 579  
 GRAFCET, 456  
 GrafetVIEW, 490–492  
 Graftek, 574  
 Graphical user interface (GUI),  
     12, 219  
 Graphs, 464–465, 556–572  
     with bivariate data, 563–565  
     intensity charts as, 571–572

Graphs (*cont.*)  
   with multivariate data, 565–569  
   3D, 570–571  
   with waveform and Cartesian data, 558–563  
**Ground**, 284–286  
**Ground loops**, 286–287  
**Ground return (low side)**, 517  
**Grounding**, 284–291  
   of differential/single-ended connections, 288–291  
   electromagnetic fields and, 286–288  
   error sources in, 288  
   ground, 284–286  
**Ground-referenced analog inputs**, 294  
**Gruggett, Lynda**, 156  
**GUI** (*see* Graphical user interface)

## H

**Hall Effect devices**, 516  
**Handle blocks**, 132  
**Hard real time**, 378  
**Hardware**:  
   GPIB issues with, 253–254  
   instrument driver issues with, 252, 254  
   I/O, 191–192  
   for LabVIEW Embedded, 422  
   for LabVIEW RT, 379–382  
   for physics applications, 514–520  
     CAMAC, 518  
     FASTBUS, 519  
     NIM module, 519–520  
     for signal conditioning, 514–518  
     VME bus, 519  
     VXI, 519  
   remote I/O, 302  
   serial instruments issues with, 252–253  
   for signal conditioning, 514–518  
     for current measurements, 516–517  
     for high-frequency measurements, 517–518  
     for high-voltage measurements, 514–516  
**Hardware platforms**:  
   for LabVIEW FPGA, 379, 380  
   RIO, 379, 380  
**HART**, 448  
**Help**:  
   compiled files for, 232  
   custom online, 233–234, 242  
**Help Tags**, 232  
**Hex Display**, 110  
**HFFAF (Hypervelocity Free-Flight Aerodynamic Facility)**, 546  
**Hide Control**, 236  
**Hide Front Panel Control**, 118

**High-accuracy timing**, 147–149  
**High-frequency measurements**, 517–518  
**Highlighting**, execution, 57  
**High-resolution timing**, 147–149  
**High-side**, of power source, 517  
**High-voltage systems**, 514–516  
**HighwayView**, 244, 460–461  
**HIST Data to Text File**, 505  
**HIST package**, 503–505  
**Histograms**, 506  
**Historical trends**, 499, 502–503  
**Hit (reset)**, 458  
**Honeywell-Measurex Corporation**, 35  
**HP Interface Bus (HP-IB)**, 247  
**HP-IB (HP Interface Bus)**, 247  
**HTML** (*see* Hypertext Markup Language)  
**HyperDrives**, 16  
**Hypertext Markup Language (HTML)**, 232  
   for document distribution, 242  
   printing to, 236  
**Hypervelocity Free-Flight Aerodynamic Facility (HFFAF)**, 546  
**Hysteresis**, 279  
**Hysteresis On/Off Controller**, 493

## I

**I16 (2-byte integer) numeric types**, 115  
**Iconic programming**, 33  
**Icons**, 49–50  
**IEEE (Institute of Electrical and Electronics Engineers)**, 247  
**IEEE 488**, 247, 253  
**IEEE 488.2**, 248  
**IEEE-488 bus**, 9  
**IFFT (Inverse Fast Fourier Transform)**, 281  
**Ignore previous input**, 157  
**IIR filters**, 347–348  
**Illustrator (Adobe)**, 237  
**Image encoding**, 573  
**Image handles**, 578–579  
**Image Source**, 581  
**Images**, 573  
**ImageToArray (IMAQ)**, 583  
**Imaging**, 555, 572–586  
   computer configuration for, 575  
   with IMAQ Vision, 577–585  
     display in, 583  
     example, 579–582  
     files in, 579  
     image handles of, 578–579  
     intensity graphs and, 583–585  
     interactive windows in, 579  
     system requirements for, 574–577  
     video I/O devices for, 575–577

**IMAGING ACquisition**, 574 (*See also* IMAQ Vision)  
**IMAQ tools** (*see specific types*, e.g.: Morphology)  
**IMAQ Vision**, 577–585  
   display in, 583  
   example, 579–582  
   files in, 579  
   image handles of, 578–579  
   intensity graphs and, 583–585  
   interactive windows in, 579  
**Importing timing data**, 146–147  
**Impulse response**, 309  
**In Range and Coerce Function**, 205, 269, 434  
**Incremental encoders**, 523  
**Independent parallel loops**, 89–90  
**Index Array**, 115, 486–487  
**Indicator events**, 82  
**Inductance**, 286  
**Inductive coupling**, 286  
**Industrial standards (process control applications)**, 438–443  
   for drawing and design, 442–443  
   for piping and instrument diagrams/symbols, 439–442  
**Infinite loops**, 65, 105  
**Info string**, 80  
**Informative alarms** (*see* Alarms)  
**Init (IMAQ)**, 577, 578  
**Initialization**, 179  
   of arrays, 117–119  
   of boolean control, 478  
   sequential control problems with, 489–490  
**Initialization VIs**, 267–268  
**Initialize and then loop**, 87–89  
**Initialize Array function**, 117, 119  
**Inline C Node**, 430  
**Implaceness**, 20  
**Input**:  
   characteristics of, 534  
   count, 173  
   data, 179  
   delimiter, 111, 174  
   ignoring, 157  
   for length, 117  
   previous, 157  
   sound, 587–588  
   type, 179, 181  
**Input offset current**, 296  
**Input scanners**, 476–477  
**Input signals**, 294–298  
   AC signals, 297  
   digital inputs, 297–298  
   floating analog inputs, 294–295  
   ground-referenced analog inputs, 294  
   thermocouples, 295–296  
**Input/output blocks (IOBs)**, external, 407  
**input/output (I/O) signals**:  
   circular buffered, 374–375

continuous control values for, 496–497  
 drivers for, 429–431  
 Error, 64  
 error flow control, 261–265  
 handler, 476  
 hardware specifications for, 191–192  
 imaging devices, 575–577  
 in LabVIEW Embedded, 429–431  
 subsystems for, 299–303, 463  
**Insert Into Array**, 117  
 Institute of Electrical and Electronics Engineers (IEEE), 247  
 Instrument data sheets, 300  
 Instrument Driver Advisory Board (LabVIEW), 259  
 Instrument Driver Finder, 244  
*Instrument Driver Guidelines* (NI), 245  
 Instrument Driver Project Wizard, 243, 259  
 Instrument drivers, 243–274  
 close VIs and, 272  
 communication standards for, 245–249  
 communication with, 251–256  
 configuration VIs and, 268–269  
 Data VIs and, 270–271  
 defined, 243  
 development of, 257–259  
 documentation of, 273–274  
 for error I/O flow control, 261–265  
 familiarization with, 249–250  
 functions of, 250–251  
 GPIB instruments, 247–249  
 hardware/wiring issues of, 252, 254  
 HighwayView and, 244  
 initialization VIs and, 267–268  
 modularity achieved with, 265–266  
 obtaining, 243–245  
 organizing with, 266–267  
 plug-and-play, 259–260  
 protocols/basic message passing of, 254–256  
 serial instruments, 245–247  
 utility VIs and, 271–272  
 Instrument I/O Assistant, 251–252  
 Instrument loop diagrams, 442  
 Instrument setups, 271, 272  
 “An Instrument That Isn’t Really” (Michael Santori), 8  
 Instrumentation:  
 amplifier for, 289  
 real, 5–6  
 virtual, 4–6  
 Instrumentation, Systems, and Automation Society (ISA), 438  
 Instrumented debugging, 421  
 Integers, 104, 115

Integral nonlinearity, 312  
 Integral responses, 498  
**Integration**, 444  
 Intensity charts, 571–572  
 Intensity graphs, 564, 583–585  
 Interactive editors, 355–357  
 Interactive windows (IMAQ Vision), 579  
 Interchannel delay, 313  
 Interface boards, 149  
 Interface names, 578  
 Interfacing, 457  
 Internal timing, 498–499  
 Internal triggering, 317  
 Internet Protocol (IP), 482  
 Interrupt Service Routines (ISRs), 434–435  
 Interrupt-driven programming (LabVIEW Embedded), 434–435  
**Interval Timer VI**, 72, 481  
 Intervals, timing, 139–140  
 Invalid enums, 134  
 Inverse Fast Fourier Transform (IFFT), 281  
 Inversion, priority, 388  
 Invoke nodes, 471  
 “Invoking a method,” 471  
 I/O (*see* Input/output)  
 IOBs (input/output blocks), 407  
 Ion beam intensity mapper application, 524–527  
 IP (Internet Protocol), 482  
 IRIG interface boards, 149  
 ISA (Instrumentation, Systems, and Automation Society), 438  
 Isolation amplifiers, 291, 515  
 Isothermal, 296  
 ISRs (Interrupt Service Routines), 434–435  
 Iteration control, 540  
 Iteration input, 72  
 Iterative design, 196

**J**

Jellum, Roger, 34  
**Jitter**, 537  
 Join Number, 132, 585  
 Joint time-frequency (JTF) domain, 281  
 JTAG emulator/debugger, 421  
 JTF (joint time-frequency) domain, 281

**K**

Karlsen, Corrie, 473  
 Keywords, 232  
 Kinetic Systems, 524  
 Kodosky, Jeff, 9–17, 19, 20, 22–24, 28, 82

**L**

**LabDriver VIs**, 20  
**Laboratory Virtual Instrument Engineering Workbench** (*see under LabVIEW*)  
**LabVIEW** (Laboratory Virtual Instrument Engineering Workbench) (general):  
 benchmark products created with, 34–36  
 challenges during development of, 16–18  
 desktop vs. FPGA use of, 406–414  
 development of, 15–18  
 future of, 32  
 on Macintosh computers, 12, 15, 19–21  
 multitasking in, 41–42  
 multithreading in, 42–43  
 origin of, 8–9  
 for PC, 15  
 portability of, 23–24  
 programming technique of, 12–15, 39–40  
 release of, 19  
 screen images of, 236–237  
 software products influenced by, 32–34  
 use of, 7–8, 37  
 virtual instrument model and, 11–12  
 vision of, 9–10  
*(See also specific applications, e.g.: LabVIEW Embedded)*

LabVIEW 1.2, 20–21

LabVIEW 2, 22–24

LabVIEW 3, 24, 25, 479

LabVIEW 4, 25–26

LabVIEW 5, 26–28, 390

LabVIEW 6, 29, 30

LabVIEW 7, 29, 30

LabVIEW 8, 31

Instrument Driver Project Wizard in, 243

Sequence structures in, 62

time functions in, 145–146

LabVIEW Application Builder, 25, 26

LabVIEW C code generator, 419, 420

LabVIEW C Runtime Library, 422–423

LabVIEW Certifications (*see specific types, e.g.: Certified LabVIEW Developer*)

LabVIEW DAQ Library, 297

LabVIEW Data Storage (Application Note AN154), 132

LabVIEW Development Guidelines, 221

LabVIEW Embedded, 417–435

best practices for, 431–434

LabVIEW Embedded (*cont.*)  
 C code and, 422–424  
 Embedded Project Manager for, 424–425  
 hardware requirements/  
 recommendations for, 422  
 history of, 417–419  
 interrupt-driven programming in, 434–435  
 I/O drivers incorporated into, 429–431  
 LabVIEW Embedded Development Module, 419  
 LabVIEW runtime library and, 422–423  
 plug-in VIs in, 425–429  
 steps in, 419–421  
 targets and, 421, 435  
 LabVIEW Embedded Development Module, 419  
 LabVIEW FPGA (field-programmable gate array), 377–391  
 application development for, 406–414  
 clocked execution and, 411–413  
 CompactRio and, 404–405  
 CompactVision system and, 405  
 compiling of, 406–408  
 debugging in, 408  
 parallelism in, 413, 414  
 pipelining in, 413–415  
 plug-in cards of, 403, 404  
 RIO hardware platforms for, 379, 380  
 synchronization and, 405, 408–411  
*LabVIEW GUI* (John Ritter), 44  
 LabVIEW Instrument Driver Advisory Board, 259  
 LabVIEW MathScript, 31  
 LabVIEW PDA Module, 419  
*LabVIEW Performance and Memory Management* (see Application Note 168 (*LabVIEW Performance and Memory Management*))  
 LabVIEW RT, 28–29, 377–399  
 communications and, 398–399  
 defining, 377–379  
 hardware for, 379–382  
 scheduling and, 395–398  
 software design for, 382–394  
     context switching in, 393–394  
     for measuring performance, 383–388  
     multithreading/multitasking in, 389–391  
     shared resources for, 388–389  
     VI organization in, 391–393  
 LabVIEW runtime library, 422–423  
 LabVIEW SCADA, 479

LabVIEW Signal Processing Toolkit, 281, 282  
 LabVIEW sketching, 202–203  
 LabVIEW Technical Notes, 119  
     (See also specific types, e.g.: *Managing Large Data Sets in LabVIEW*)  
 LabWindows CVI, 33  
 Ladder logic, 456, 486  
 Lag (delay), 444, 492  
 Langmuir probe, 527  
 Latch modes, 77  
 Latching, 508  
 Latency, 378, 379  
 Lawrence Livermore National Laboratory, 34  
 LCD panels, 194  
 Lead, 492  
 Lead/Lag VI, 492  
 Leak resistors, 295  
 LEDs (light-emitting diodes), 298  
 Length input, 117  
 Libraries:  
     C Runtime Library, 422–423  
     Call Libraries, 59, 434  
     DAQ Library, 297  
     for digital timing, 149  
     DLLs, 29, 59, 149  
     LabVIEW runtime library, 422–423  
     RDTSC library, 383  
     shared, 59  
     VTL, 257  
 Light-emitting diodes (LEDs), 298  
 Limiting, derivative, 497  
 Line feed, 111  
 Linear Technology, 297  
 Linearity, 312  
 Link list, 178  
 Linked code, 420  
 Linking, 423  
 Links, 45, 178  
 List, of links, 178  
 List Folder, 169  
 Local variables, 24, 75–78, 80  
     debugging of, 80  
     in desktop and FPGA LabVIEW use, 406  
     finding, 80  
     global variables vs., 431  
     sequence, 61  
 Lock-in amplifiers, 545  
 Locking, 83, 84, 250  
 Log always, 215  
 Log At Completion, 182  
 Log command, 182  
 Log on change, 215  
 Logic functions, boolean, 486–487  
 Loop tunnels, 431  
 Looping, 64–71  
     aborting, 67  
     constants and, 431  
     dispatcher loop, 470  
     independent parallel loops, 89–90

infinite, 65, 105  
 initialize and then loop, 87–89  
 For Loops, 64, 66–67  
 pretest While Loops, 65  
 shift registers for, 67–71  
 stops for, 65, 66  
 Timed Loop with Frames, 140  
 Timed Loops, 138, 140, 395  
 While Loops, 64–66  
 Loops, 13  
 Loops per second, 498  
 Low side (ground return), 517  
 Low-pass filter, 307  
 Low-speed DAQ, 370–372  
 LT1088 RMS to DC converter, 297

## M

MacCrisken, Jack, 9, 16, 20, 23, 214  
 Machine-dependent (manager) layer, 24  
 Macintosh computers, 15–17  
     LabVIEW development on, 12, 15  
     sound files on, 589  
     timers and, 137  
 Macintosh II, LabVIEW development on, 19–21  
 Macintosh Plus, 17  
 MacPaint, 12, 14  
 Magnetic shielding, 286  
 Mailboxes, files as, 484  
 Make Current Value Default, 118, 206  
 Manager (machine-dependent) layer, 24  
*Managing Large Data Sets in LabVIEW* (LabVIEW Technical Notes), 119  
 Manipulated variable, 444  
 Man-machine interfaces (MMI), 463–475  
     design of, 44  
     display hierarchy in, 469–473  
     front-panel item handling in, 474–475  
     standards for, 438  
     techniques for, 473–474  
 Mantissa, 128  
 Manual control, 444  
 Manual save and recall, 367  
 Maps, 565  
 Match First String, 108  
 Match Pattern, 100–101, 107, 108  
 MathScript (LabVIEW), 31  
 Matrix (2D array), 112  
 Median filters, 348  
 Medium-speed DAQ, 373–375  
 Mega-applications, 22  
 Memory:  
     allocation of, 431  
     diagnostics for, 552  
     optimization of, 547–553  
     problems with, during LabVIEW development, 17, 20

- segmented, 535  
state, 70
- Memory manager, 121
- Memory Usage, 119–122, 552
- Menu-driven systems, editors for, 360–361
- Message passing:  
of GPIB instruments, 254  
of instrument drivers, 254–256  
of serial instruments, 254–256
- Metainformation, 176
- Metal-oxide varistors (MOVs), 516
- Metric Systems, 570
- Microcontrollers, 455
- Microsoft Excel, epoch time in, 147
- Microsoft Windows 3.0, 15, 23
- Microstepping, 521
- MightySat II.1* (Air Force Research Lab), 34
- Milliseconds (ms) timing, 138, 139
- MMI (*see* Man-machine interfaces)
- MMI G Wizard, 469
- Modbus protocol, 457
- Mode control, 216, 486
- Model-based controllers, 496
- Modem, null, 247, 253
- Modicon, 457
- Modularity, 198–200, 265–266
- Moore Products, 292
- Morphological transformations, 582
- Morphology (IMAQ), 582
- Motion control systems, 520–523
- Motion Toolbox, 214
- Motorola 68000 microprocessors, 15
- Move, 169
- Moving averager, 308
- Moving averages, 348–350
- MOVs (metal-oxide varistors), 516
- ms time out, 156
- Ms (milliseconds) timing, 138, 139
- Multidrop systems, 246
- Multimedia, 555
- Multimeter, digital, 250
- Multiple state machines, 488
- Multiple-loop controllers, 462
- Multiple-record mode, 535
- Multiplexing:  
ADCs vs., 313  
in signal conditioning, 293  
timing skew and, 312–313
- Multipulse data, 539–543
- Multitasking, 41–42, 389  
cooperative, 41–42  
preemptive, 145  
in RT software design, 389–391
- Multithreaded operating systems, 164
- Multithreading, 27, 42–43, 145, 389–391
- MultiThreshold (IMAQ), 581
- Multivariate data, on graphs, 565–569
- Mutex (Mutually exclusive), 163
- Mutliple time bases, 535
- Mutually exclusive (mutex), 163, 388
- MXProLine, 35
- MyGraphData, 483
- N**
- Namespace conflicts, 161
- Naming:  
of cursors, 567  
of tags, 439–441, 467  
unique, 71
- NaN (*see* Not-a-number)
- Nanosecond waveforms, 542
- NASA:  
Ames Research Center's HFFAF, 546  
*New Millennium Deep Space 1*, 34
- National Electrical Code, 285, 443
- Naval Research Lab Coriolis spacecraft, 34
- Needs analysis, user, 186–187
- Negative feedback, 444
- Network connections, 482–484
- Network-Published Shared Variable, 398
- New Millennium Deep Space 1* (NASA), 34
- Newport Corporation, 523
- NI Spy, 256
- NI-FBUS host software, 448
- NI-IMAQ, 577
- NIM (Nuclear Instrumentation Manufacturers) module, 519–520
- NI-PSP (Publish-Subscribe Protocol), 398
- NI-SCOPE driver, 533
- Nodes, 38, 202  
call by reference, 471  
CINs, 58–59  
property, 45–47, 80, 194, 471, 474
- Noise:  
dithering of, 318  
elimination of, 284  
in signal sampling, 317–319
- Nonlinearity, integral, 312
- Normal-mode signal, 289, 485
- Not A Number/Path/Refnum, 105, 207
- Not-a-number (NaN), 105, 207
- Notification VI, 159
- Notifiers, for synchronization, 158–160
- Notify events, 83, 153
- NuBus architecture, 20
- Nuclear Instrumentation Manufacturers (NIM) module, 519–520
- Null modem, 247, 253
- Null (zero) terminators, 134
- NuLogic, 522
- Number of Cycles control, 531
- Numbers and numbering:  
incorrect, 96  
of rows, 173  
state, 94
- Numeric types, 104–105  
clusters and, 115  
DBL, 115  
I16, 115  
SGL, 115
- Nyquist rate/criterion, 305–307
- O**
- Object files, 423
- Object oriented programming (OOP), 22
- Occurrences, 155–158
- OCDIComments, 428
- OCX (oven-controlled crystal oscillator), 149
- Offset, 180, 183
- Offset errors, 311
- Ohm's Law, 516
- OLE for Process Control (OPC), 456
- Omega Engineering Temperature Handbook, 296
- On-chip debugging, 421
- One-dimensional (1D) string array, 69, 111
- Online help, custom (*see* Custom online help)
- On/off controllers, 444
- OOP (object oriented programming), 22
- OPC (OLE for Process Control), 456
- OPC-based PLC (example), 458–460
- Open standards, 450
- Open VI Reference node, 471
- Open/Create/Replace File, 169, 170, 182
- Opening, 582
- Operating systems,  
multithreaded, 164
- Operation mode, 170
- Operators, notification alarms for, 511–512
- Opto-22, 302
- Output (out) control, 467
- Output data:  
analog, 299  
data distribution of, 477–479  
manipulation of, 444–447  
in process control applications, 444–447  
for sound, 588
- Oven-controlled crystal oscillator (OCX), 149
- Overflow, 104, 310
- Oversampling, 308, 319, 344

## P

- P (proportional) controllers, 444  
 PACs (programmable automation controllers), 377  
 Page, 121  
 Parallel loops, independent, 89–90  
 Parallelism:  
     in desktop and FPGA LabVIEW  
         use, 406  
     in LabVIEW FPGA, 413, 414  
 Parametric Graph, 3D, 570  
 Parametric plots, 560  
 Parasitic detection, 287  
 Pareto analysis, 506  
 Parker, Jeff, 20, 570, 571  
 Parker Compumotor, 214  
 Parsing, 107–109  
 Path control, 168  
 Path To String, 168  
 Pattern matching (IMAQ), 580  
 Pause button, 55  
 PCI 7041/6040E, 380, 381  
 PCI bus interface, 519  
 PCs (*see* Personal computers)  
 PDA Module, 419  
 PDA Module (LabVIEW), 419  
 PDAs (Personal Digital Assistants), 417–419  
 Peak voltage, 240  
 Peer-to-peer communications, 453  
 Performance:  
     continuous control limitations  
         on, 498–499  
     RT software design  
         measurements of, 383–388  
         strings affecting, 476  
 Performance and Disk items, 552  
*Performance and Memory Management* (Application Note 168), 122  
 Performance checking, 216–217  
 PERL language, 28  
 Personal computers (PCs), 15  
     control system architectures  
         with, 451–452  
     desktop, 379, 381  
 Personal Digital Assistants (PDAs), 417–419  
 PGIA (programmable-gain instrumentation amplifier), 314  
 PharLap TNT Embedded, 28  
 Phase response, 309  
 Phase-sensitive detectors (PSDs), 545  
 Physics applications, 513–553  
     data quantity in, 546–553  
         memory optimization for, 547–553  
         reduction of, 546–547  
     for field diagnostics, 520–527  
         ion beam intensity mapper application, 524–527  
     motion control systems for, 520–523  
         reading encoders and, 523  
     hardware for, 514–520  
         CAMAC, 518  
         FASTBUS, 519  
         NIM module, 519–520  
         for signal conditioning, 514–518  
         VME bus, 519  
         VXI, 519  
     for plasma diagnostics, 527–532  
     pulse and transient phenomena  
         in, 533–546  
         DSOs for, 536–539  
         multipulse data, 539–543  
         signal recovery for, 543–546  
         transient digitizers for, 533–536  
 PI (proportional-integral) controllers, 445  
 PIC VIs, 497–498  
 Pick Line, 106  
 PICT objects, 236  
 Pict Ring, 193, 474  
 Picture Controls, 194, 567, 583  
 Picture in–Picture out, 569  
 Pictures, 567  
 PID (algorithm), 492  
 PID (Gain Schedule), 492, 495  
 P&ID (piping instrument diagrams and symbols), 439–442  
 PID Control Toolkit, 446, 485, 492, 497  
 PID (proportional-integral-derivative) controllers, 444  
 PID tuning schedule, 495–496  
 PID with Autotuning, 492  
 Pipelining (LabVIEW FPGA), 413–415  
 Piping instrument diagrams and symbols (P&ID), 439–442  
 Pixels, 573  
 Plagiarizing, 59–60  
 Plasma diagnostics, 527–532  
 PLCs (*see* Programmable logic controllers)  
 Plot(s):  
     Contour, 564  
     defined, 560, 562  
     parametric, 560  
     surface, 563, 570  
     waterfall, 563, 565  
     wire-frame, 570  
 Plug-and-play instrument drivers, 259–260, 266  
 Plug-in boards, 253–254  
 Plug-in cards, 403, 404  
 Plug-in VIs, 425–429  
     code generation for, 426–429  
     Target\_OnSelect, 426  
 Points in Graph Window, 501–502  
 Polling, 152–153  
 Polymorphic data, 68  
 Polymorphic VIs, 50  
 Polymorphism, 103, 179, 559  
 Pop-up editors, 357–360  
 Pop-up menus, 232, 273  
 Portability, 23–24  
 Portable Document Format (PDF), 242  
 porting), 23  
 Ports:  
     addition of, 246–247  
     for serial instruments, 246–247  
 Pos offset, 183  
 Position Meas Quad Encoder VI, 523  
 Positive feedback, 444  
 Postrun analysis for, 326–336  
     in ASCII text format, 328–329  
     configuration information for, 333–335  
     in custom binary formats, 329–331  
     database for, 335–336  
     in datalog file format, 326–327  
     Datalog File Handler VI for, 327–328  
     direct links in, 331–332  
     timestamps for, 332–333  
 PostScript printing, 237  
 Posttrigger data, 534  
 Potential (volts) controls, 78  
 Power (watts) controls, 78  
 Power spectrum, 306  
 P-picker, 546  
 Precision, 104  
 Precision menu, 132  
 Predictive controllers, 446, 496  
 Preemptive multitasking, 145  
 Preemptive scheduling, 391  
 Prepare VI, 428  
 Preston (company), 292  
 Pretest While Loops, 65  
 Pretrigger data, 534  
 Primary elements, 447  
 Print command, 235, 236  
 Print Options, 236  
 Print Window, 236  
 Printing:  
     of documentation, 235–238  
     functions for, 269  
     to HTML, 236  
     with PostScript, 237  
 Priority, 91, 391  
 Priority boosting, 389  
 Priority inversion, 388  
 Probe characteristic, 527  
 Probe tools, 55  
 Problem definition, 186–191  
 Process control applications, 437–512  
     alarms and, 506–512  
         alarm handler for, 508–511  
         for operator notification, 511–512  
     for continuous control, 492–499  
         control strategy for, 493–496

I/O values in, 496–497  
 PIC VIs and, 497–498  
 timing/performance  
     limitations on, 498–499  
 control system architectures for,  
     449–455  
 DCS, 449–451  
 with personal computers,  
     451–452  
 SCADA, 451  
 selection of, 453–455  
 data distribution in, 475–486  
 with display VIs, 479–482  
 with input scanners, 476–477  
 of output data, 477–479  
 real-time process control  
     databases for, 484–485  
     using network connections,  
         482–484  
     validation and, 485–486  
 industrial standards for, 438–443  
 for drawing and design,  
     442–443  
 for piping and instrument  
     diagrams/symbols, 439–442  
 man-machine interfaces and,  
     463–475  
 display hierarchy in, 469–473  
 front-panel item handling in,  
     474–475  
 techniques for, 473–474  
 output manipulation in, 444–447  
 process signals in, 447–449  
 for sequential control, 486–492  
     boolean logic functions in,  
         486–487  
 GrafetVIEW and, 490–492  
 initialization problems in,  
     489–490  
 state machine architecture for,  
     487–488  
 with smart controllers, 452,  
     455–463  
 I/O subsystems, 463  
 PLCs, 455–461  
 trending and, 499–506  
     HIST package and, 503–505  
     historical trends, 502–503  
     real-time trends, 499–502  
     SPC and, 505–506  
**Process Control Systems**  
 (Greg Shinsky), 446  
 Process mimic displays, 464  
 Process signals, 447–449  
 Process variable, 444  
 Processing, image, 573  
 Producer, 160, 210  
 Producer events, 209  
 Profile Memory Usage, 552  
 Profile VIs, 216  
 Profiling, 552  
 Program architecture, 87  
 Programmable automation  
     controllers (PACs), 377

*Programmable Controllers: Theory  
 and Implementation*  
 (ISA), 456  
 Programmable logic controllers  
     (PLCs), 377, 455–461  
     advantages of, 456–457  
     from Allen-Bradley, 244  
     communications techniques of,  
         457–458  
     HighwayView, 460–461  
     OPC-based example of, 458–460  
     sequential control systems  
         based on, 447  
 Programmable-gain  
     instrumentation amplifier  
     (PGIA), 314  
 Programming:  
     in CLAD, 219  
     with dataflow, 38–39  
     iconic, 33  
     of LabVIEW, 12–15, 39–40  
     by plagiarizing, 59–60  
 Programs, robust, 204  
 Projectors, RGB, 194  
 Property nodes, 45–47, 194,  
     471, 474  
     debugging of, 80  
     defined, 45  
 Proportional band, 497  
 Proportional (P) controllers, 444  
 Proportional-integral (PI)  
     controllers, 445  
 Proportional-integral-derivative  
     (PID) controllers, 444  
 Protocols:  
     of GPIB instruments, 254  
     of instrument drivers, 254–256  
     of serial instruments, 254–256  
 Prototyping:  
     rapid, 196  
     of user interface, 192–196  
 PSDs (phase-sensitive  
     detectors), 545  
 Pseudocoding, 203–204  
 Publishing, 398  
 Publish-Subscribe Protocol  
     (NI-PSP), 398  
 Pull-up resistors, 297  
 Pulse, 108, 403  
 Pulse and transient phenomena,  
     533–546  
     DSOs for, 536–539  
     multipulse data, 539–543  
     signal recovery for, 543–546  
     transient digitizers for, 533–536  
 Pulse-width modulator (PWM)  
     encoder, 404  
 PXI system, 381–382, 405

**Q**

Quad int, 146  
 Quadrature encoders, 523

Quantization, 309  
 Queue(s), 91, 160–161, 479  
     defined, 160  
     for synchronization, 160–161  
     use of, 161  
 Queued Message Handler, 98–99,  
     161, 209  
 Quick start, 239, 241  
 QuickDraw, 15  
 QuVIEW, 321

**R**

Race conditions, 73, 91, 163  
 Radio-frequency interference  
     (RFI), 287  
 Ramp Pattern VI, 531  
 Random accessibility, 177  
 Range, 104, 310  
 Range Finder VI, 205  
 Rapid prototyping, 196  
 Rate, of commands, 270  
 RDA (see Remote data acquisition)  
 RDTSC, 383  
 Read Datalog, 181  
 Read File, 63, 64  
 Read From Measurement  
     File VI, 176  
 Read From Spreadsheet File, 173  
 Read From Text File, 60  
 Read function (DataSocket), 483  
 Read Lines From File, 174  
 Read Mark, 171  
 Read Text function, 173  
 Read the fine manual (RTFM), 548  
 Read The Manual (RTM), 243  
 Reading:  
     binary files, 179–181  
     datalog files, 182–183  
     text files, 172–174  
 Ready (thread), 391  
 Real instrumentation, 5–6  
 Real knobs, 536  
 Real time (RT), 377–379 (See also  
     LabVIEW RT)  
 Real time analysis, 337–343  
     with continuous data, 339–340  
     display performance for, 342–343  
     with single-shot data, 338–339  
     speed of, 340–341  
     volume of data for, 341–342  
 Real (floating-point) value, 104  
 Real-time databases, 475, 476,  
     484–485  
 Real-time operating system  
     (RTOS), 28, 377, 378  
 Real-time trends, 499–502  
 Real-time xy strip chart, 562  
 Rebooting, 250, 454  
 Recipe operation, 446  
 Reconfigurable I/O (RIO), 403  
 Reconstruction filters, 315  
 Record (Pascal), 50

Recording, 365–370  
 automatic save and recall, 367–370  
 manual save and recall, 367

Recovery, error, 261

Red File (IMAQ), 579

Red-green-blue (RGB) projectors, 194

Redundancy, 454

Reentrant, 28, 54, 162

Reference junction, 296

Reference numbers (refnums), 167  
 datalog, 181  
 file, 169

Referenced single-ended (RSE) input, 295

Refnums (*see* Reference numbers)

Region of interest (ROI), 579

Register For Events, 86

Release Semaphore VI, 164

Remote data acquisition (RDA), 303–304

Remote I/O hardware, 302

Remote sensors, 302

Remote terminal units, 451

Rendezvous VIs, 165–166

Reorder Controls in Cluster, 122

Repeat addressing, 254

Replace Array Element, 120, 479, 502

Replace Array Subset, 117

Representation, 129

Required connections, 239

Resample (IMAQ), 583

Resampling, 583

Reset Structure at Start feature, 143

Reset windup, 445

Resetting, 234, 444, 458

Resistance temperature detectors (RTDs), 293

Resistors, pull-up, 297

Resolution (bits), 309

Resolution multiplication, 523

Restoration, image, 573

Return paths, 285

Return to Caller button, 58

Reverse Array, 132

Reverse String, 132

Revision history, 234–235

RFI (radio-frequency interference), 287

RGB (red-green-blue) projectors, 194

Rhia, Stephen, 158

Rich text format (RTF) files, 236, 237

Ring and Enum control palette, 133

Ring Indicators, 130

RIO (reconfigurable I/O), 403

RIO hardware platforms (LabVIEW FPGA), 379, 380

Ritter, Dave, 44, 589

Robust programs, 204

Rogers, Steve, 20, 24

ROI (region of interest), 579

Rosemount, 448

Round-robin scheduling, 391

Route Signal VI, 385

Rows, 116, 121  
 first, 173  
 in IMAQ, 583  
 number of, 173  
 output of, 173  
 values of, 181

RS-232C, 246, 278, 303

RS-422A, 246

RS-432A, 246

RS-485A, 246

RSE (referenced single-ended) input, 295

R-series DAQ cards, 403–404

RT (real time) (*see* LabVIEW RT)

RTDs (resistance temperature detectors), 293

RTF (*see* Rich text format files)

RTFM (read the fine manual), 548

RTM (Read The Manual), 243

RTOS (*see* Real-time operating system)

Run button, 58

Running (thread), 391

## S

Safety:  
 in high-voltage system design, 515–516  
 signal conditioning for, 292–293

Safety ground, 285

Sample-and-hold (S/H) amplifier, 310

Samples, 280

Sampling theorem, 305–307

Santori, Michael, 8

Saturation, 310

Save and recall:  
 automatic, 367–370  
 manual, 367

Save button, 216

SCADA (*see* Supervisory Control and Data Acquisition)

Scaler (single-element) types, 103

Scan From File, 175

Scan From String, 108–110

Scan interval, 460

Scan Mode, 525

Scanner, 476

Scheduler queue, 140

Scheduling:  
 LabVIEW RT and, 395–398  
 preemptive, 391  
 round-robin, 391

SCPI (Standard Commands for Programmable Instruments), 248, 258

Screen captures, 237

Screen images, 236–237

ScriptBit VI, 426

ScriptCompiler, 428–429

ScriptLinker, 428, 429

SCSI (Small Computer Systems Interface) ports, 19

SCXI equipment, 292, 294, 303

Second-level triggers, 538

Seconds To Date/Time, 138, 146

Seeback effect, 288

SEG (Software Engineering Group), 244

Segmentation, images, 573

Segmented memory, 535

Select function, 432

Select Item, 123

Self-inductance, 286

Semaphores, 163  
 defined, 490  
 synchronization with, 161–165  
 used in multithreaded operating systems, 164

Sensor systems, 278

Sensors, 277–278, 302

Sequence local variables, 61

Sequence structures and sequencing, 14, 61–62  
 avoidance of, 62  
 coding schemes of, 62  
 in LabVIEW 8, 62

Sequential control, 446, 486–492  
 boolean logic functions in, 486–487  
 GrafetVIEW and, 490–492  
 initialization problems in, 489–490  
 state machine architecture for, 487–488

Sequential function charts (SFCs), 490

Serial communications (*see specific types, e.g.: RS-232C*)

Serial devices, 245  
 parameters of, 255  
 problems with, 247

Serial instruments, 245–247  
 hardware/wiring issues of, 252–253  
 ports for, 246–247  
 protocols/basic message passing of, 254–256  
 RS-series of, 246  
 troubleshooting of, 247

Serial Number, 147

Serial Port functions, 255, 511

Serial Port Init VI, 247

Server, 90

Service requests (SRQs), 270

Servo motors, 520

Set Cluster Size, 135

Set File Position Function, 179

Set Occurrence, 155

Set Value, 71

- Set Waveform Attribute, 127  
 Setpoint (SP), 466, 467  
 Settling time, 292, 313–315  
 SFCs (sequential function charts), 490  
 SGL (single-precision floating point) numeric types, 115  
 S/H (sample-and-hold) amplifier, 310  
 Shah, Darshan, 28, 29  
 Shannon sampling theorem, 305–307  
 Shared libraries, 59, 149  
 Shared resources:  
     defined, 388  
     for RT software design, 388–389  
     synchronization of, 161–165  
 Shared Variable Engine (SVE), 398  
 Sheldon Instruments, 321  
 Shielding, 286 (*See also* Grounding)  
 Shift registers:  
     for large arrays, 431  
     for looping, 67–71  
     uninitialized, 69–71  
 Shinskey, Greg, 446  
 Short-circuit (crowbar), 516  
 Show Buffer Allocations, 121  
 Show VI Hierarchy, 201  
 Shunting, 516–517  
 SI Clear, 587  
 SI Config, 587  
 SI Read, 587  
 SI Stop, 587  
 Sign bit, 315, 316  
 Signal(s), 275–304  
     actuators, 278–279  
     ADCs and, 275  
     analog, 305  
     balanced, 51, 290  
     categories of, 279–283  
     common-mode, 289  
     connections between, 284–304  
         amplifier use for, 291–294  
         grounding/shielding of, 284–291  
     of input signals, 294–298  
     with networking, 303–304  
     of output signals, 298–299  
 controller, 447  
 EOI, 254  
 input, 294–298  
     AC signals, 297  
     digital inputs, 297–298  
     floating analog inputs, 294–295  
     ground-referenced analog inputs, 294  
     thermocouples, 295–296  
 I/O subsystem for, 299–303  
 normal-mode, 289  
 origins of, 275–283  
 sensors, 277–278  
 transducers, 276–277  
 Signal bandwidth, 343–344  
 Signal common, 285–286  
 Signal conditioning, 284, 291  
     hardware for, 514–518  
         for current measurements, 516–517  
         for high-frequency measurements, 517–518  
         for high-voltage measurements, 514–516  
     multiplexing in, 293  
     for safety, 292–293  
 Signal Processing Toolkit, 281, 282  
 Signal recovery, 543–546  
 Signal Recovery By BP Filter, 544  
 Signal sampling, 305–321  
     analog-to-digital converters for, 309–314  
     averaging in, 308–309  
     coding schemes for, 315–316  
     digital-to-analog converters for, 314–315  
     filtering in, 307–309  
     noise in, 317–319  
     sampling theorem, 305–307  
     throughput and, 319–321  
     triggering and, 316–317  
 Signal-to-noise ration (SNR), 291  
 Signed integers, 104  
 Significant figures, 105  
 Simple diagram help, 239  
 Simple Error Handler, 171, 209, 210, 212, 532  
 Simplicity, 44  
 Simulation mode, 485  
 Sine Waveform VIs, 125  
 Single-element (scaler) types, 103  
 Single-ended connections, 288–291  
 Single-loop controllers (SLCs), 453  
 Single-precision floating point (SGL) numeric types, 115, 549  
 Single-shot data:  
     continuous data vs., 338  
     real time analysis with, 338–339  
 Single-stepping, 55–57  
 Sins (*see* Code Interface Nodes)  
 Site machines, 94–98  
 6B Series (*see* Analog Devices 6B Series)  
 16550-compatible UART, 246  
 sketching (LabVIEW), 202–203  
 Skew, 537  
 Skin effect, 286  
 “Skip if busy,” 393  
 SLCs (single-loop controllers), 453  
 Slew rate, 314–315  
 Slices, 401, 407  
 Slider controls, 78, 466  
 Small Computer Systems Interface (SCSI) ports, 19  
 Smart controllers, 452, 455–463  
     I/O subsystems, 463  
     programmable logic controllers, 455–461  
         advantages of, 456–457  
     communications techniques of, 457–458  
     HighwayView, 460–461  
     OPC-based example of, 458–460  
     single-loop controllers, 461–463  
 Snapshot button, 216  
 Snd Read Wave File, 588–589  
 Snd Read Waveform, 587  
 Snd Write Wave File, 588–589  
 Snd Write Waveform, 588  
 Snider, Dan, 214  
 SNR (signal-to-noise ration), 291  
 SO Pause, 588  
 SO Wait, 588  
 SO Write, 588  
 Soft real time, 378, 379  
 Software constructs, in CLAD, 219  
 Software design (RT), 382–394  
     context switching in, 393–394  
     for measuring performance, 383–388  
     multithreading/multitasking in, 389–391  
     shared resources for, 388–389  
     VI organization in, 391–393  
 Software Engineering Group (SEG), 244  
 Software-generated triggers, 317  
 Sound files, 588–589  
 Sound I/O, 586–589  
     DAQ for, 586–587  
     files for, 588–589  
     functions of, 587  
     input, 587–588  
     output, 588  
 SoundBlaster, 587  
 Source boolean (string), 262–263  
 Source pins, 523  
 Sources (timing), 141–143  
 SP (*see* Setpoint)  
 Sp low, 497  
 SP-50, 448  
 Span (range), 497  
 Spatial filtering, 582  
 SPC (statistical process control), 505–506  
 Specifications definition, 187–189  
 Spectrogram, 281  
 Spectrum Astro, Inc., 34  
 Speed, of real time analysis, 340–341  
 Speed Tester, 191  
 Spike, 313  
 Split Number, 132, 585  
 Spreadsheet programs, 12  
 Spreadsheet String To Array specifier, 111  
 Spreadsheets, strings in, 110–114  
 SQC (*see* Statistical process control)  
 SRQs (service requests), 270  
 Standard Commands for Programmable Instruments (SCPI), 248, 258

Start button, 216  
 Start frame, 488  
 Start-up management, 496  
 State diagrams, 13, 97  
 State machines, 70, 487–488  
 State memory, 70  
 State number, 94  
 State variable analysis, 446  
 Static editors, 353–355  
 Static links, 45  
 Statistical process control (SPC), 505–506  
 Statistical Process Control Tools, 505  
 Statistical quality control (*see* Statistical process control)  
 Status boolean, 262  
 Status displays, for editors, 360–361  
 Step Into, 56  
 Step Out Of, 56  
 Step Over, 56  
 Step-and-measure experiments, 520–527  
     ion beam intensity mapper application, 524–527  
     motion control systems for, 520–523  
     reading encoders and, 523  
 Stepper motors, 520  
 Stop button, 531  
 Stop switch, 77  
 Stops (looping), 65, 66  
 Store HIST Data VIAs, 504  
 Strict typedef, 51, 122, 198  
 String(s), 105–114  
     Array To Spreadsheet String, 111  
     building, 106–107  
     comparison functions for, 269  
     data, 132  
     as default, 180  
     info, 80  
     parsing, 107–109  
     performance affected by, 476  
     in spreadsheets, 110–114  
     unprintables, 110, 111  
 String Subset, 109  
 String to Byte Array, 583  
 String To Path, 168  
 Strip charts, 464–465  
 Strip Path function, 168–169  
 Struct (C), 50  
 Structured dataflow, 13  
 Structured design:  
     bottom-up, 197–198  
     top-down, 197  
 Structured types, 103  
 Style (CLD), 222  
 Sub Panel Pane events, 81–82  
 Subdiagrams, 13  
 Subroutine, 551  
 Subroutine priority, 391  
 Subscribers, 398  
 SubVI Node Setup, 58

SubVIs, 48  
     debugging with, 54–55  
     defining, 5, 48, 198  
     design of, 220  
     in desktop and FPGA LabVIEW use, 406  
     development of, 19  
 Sun SPARCstation, 24  
 Supervisory Control and Data Acquisition (SCADA), 451, 479  
 Surface plots, 563, 570  
 SurfaceVIEW, 570  
 Suspend When Called  
 SVE (Shared Variable Engine), 398  
 Swap Bytes (Swap Words), 132  
 Swap Words (Swap Bytes), 132  
 Switched capacitor filters, 292  
 Switching:  
     context, 377  
     stop, 77  
 Sync, 317  
 Synchronization, 151–166  
     of events, 153–155  
     in LabVIEW FPGA, 405, 408–411  
     notifiers for, 158–160  
     of occurrences, 155–158  
     with polling, 152–153  
     queues for, 160–161  
     with rendezvous VIs, 165–166  
     with semaphores, 161–165  
     of shared resources, 161–165  
     (*See also* Timing)  
 Synchronous amplifiers, 545

## T

T (transparent) color, 557  
 Tag Configuration Editor, 459  
 Tag names, 439–441, 467  
 Tags (I/O points), 459  
 Tanzillo, P. J., 417  
 Target Properties, 408  
 Targets, 421, 435  
 Tasks, 198  
 TCP (Transmission Control Protocol), 482  
 TDC (time-to-digital converter), 518  
 Technical Notes (LabVIEW) (*see* LabVIEW Technical Notes)  
 Tektronix, 514  
 TelevEyes, 577  
*Temperature Handbook* (Omega Engineering), 296  
 Terminal descriptions, 240–241  
 Termination character, 268  
 Terminators, null, 134  
 Test and verify, 217  
 Text files, 169–176  
     ASCII, 167  
     formatting to, 175–176

reading, 172–174  
     writing, 170–172  
 Thermocouples, 295–296  
 Third-level triggers, 538  
 32-bit color, 585  
 Threadconfig, 390  
 Threads, 42  
 Thread-safe, 28  
 3D Curve Graph, 570  
 3D graphs, 570–571  
 3D Parametric Graph, 570  
 Thresholding, 580  
 Throughput, 191, 319–321  
 Tick Count (ms), 138  
 Time base (clock), 316  
 Time buffered I/O, 373  
 Time critical, 389  
 Time domain, ac, 280  
 Time Has Elapsed, 222  
 Time out (ms), 156  
 Time Target(s), 222  
 Time-critical code, 434  
 Timed Loop with Frames, 140, 395  
 Timed Loops, 138, 140, 395  
 Timed Sequence Structure, 138, 140, 395  
 Timed structures, 385, 395–398  
 Timeout frame, 488  
 Timers, 137–138  
 Time-sampling, equivalent, 306  
 Timestamps, 146, 332–333  
 Time-to-digital converter (TDC), 518  
 Timing, 137–149 (*See also* Synchronization)  
     absolute timing functions, 139  
     continuous control limitations on, 498–499  
     digital, 149  
     for digital filtering, 350  
     epoch, 145, 146  
     execution/priority of, 143–145  
     external, 498, 499  
     guidelines for, 145–146  
     high-resolution/high-accuracy, 147–149  
     importing/exporting data for, 146–147  
     internal, 498–499  
     intervals for, 139–140  
     in LabVIEW 8, 145–146  
     in milliseconds, 138, 139  
     sources for, 141–143  
     structures for, 140–141  
     timers for, 137–138  
 Timing skew, 312–313  
 Tip pop-up menu, 232, 273  
 Tip strips, 25, 56–57  
 To Upper Case function, 108  
 Toolchain, 421  
 Top index (channel), 120  
 Top-down structured design, 197  
 Total Range, 525  
 Trace commands, 214–216

Trace VI, 214–217  
 Training, of users, 217  
 Transducers, 276–277, 447  
 Transfer function, 308  
 Transformations, morphological, 582  
 Transformers, current, 516  
 Transient digitizers, 518, 533–536  
     calibration of, 535–536  
     data storage/sampling in, 534–535  
     input characteristics of, 534  
     for pulse and transient phenomena, 533–536  
     triggering in, 534  
 Transient phenomena (*see* Pulse and transient phenomena)  
 Transmission Control Protocol (TCP), 482  
 Transmit lines, 247  
 Transmitters, 447  
 Transparent (T) color, 557  
 Transpose Array, 181, 584  
 Transpose option, 172  
 Transposing, 173  
 Trends and Trending, 499–506  
     displays of, 464–465, 499  
     HIST package and, 503–505  
     historical trends, 502–503  
     real-time, 499  
     real-time trends, 499–502  
     statistical process control and, 505–506  
 Trigger delays, 537  
 Trigger discriminator, 537  
 Trigger exclusion, 539  
 Trigger fan-out, 537  
 Triggering, 309  
     complex, 403  
     defined, 316  
     external, 316–317  
     first-level, 537  
     internal, 317  
     second-level, 538  
     signal sampling and, 316–317  
     third-level, 538  
     in transient digitizers, 534  
 TRMS (true RMS) converter module, 297  
 Troubleshooting serial instruments, 247  
 Truchard, Jim, 9–11, 17, 23  
 True RMS (TRMS) converter module, 297  
 TrueTime, 149  
 Tufte, Edward, 556, 557  
 Turnkey systems, 450  
 2-byte integer (I16) numeric types, 115  
 2D (two-dimensional) array (matrix), 112  
 Two-dimensional (2D) array (matrix), 112  
 Type, wiring to, 180  
 Type Cast, 585

Type casting, 130, 134–135  
 Type Definition (typedefs),  
     52–53, 198  
     clusters saved as, 51, 122  
     Control Editor and, 53  
     for data type determination, 182  
     defined, 52  
     strict, 51  
 Type descriptor, 128  
 Type input, 179, 181  
 Type setting, 129–132  
 Type specifier, 111, 472  
 Typedef, strict (*see* Strict typedef)  
 Typedefs (*see* Type Definition)

## V

Vacuum brazing, 187  
 Valid indicator, 71  
 Validation, 485–486  
 Value, 216  
 Value Change event, 86–87  
 Variables, 219 (*See also specific types, e.g.: Global variables*)

Variant data fields, 126  
 Variant To Data, 127  
 VHDL (VHSIC hardware description language), 402  
 VHSIC hardware description language (VHDL), 402

VI descriptions:  
     in documentation, 231  
     documentation of, 231–232,  
     239, 240

VI events, 81  
 VI hierarchy, 201–202

VI history:  
     documentation of, 234–235  
     of revisions, 234

VI Properties, 200, 273, 552

VI Revision History, 234

VI Server, 471–473

VI Tree, 267

Video I/O devices, 575–577

Videotape, 575

Virtual instrument model, 11–12

Virtual Instrument Software Architecture (VISA), 257, 272 (*See also under VISA*)

Virtual instruments (VIs), 4, 40–41

autonomous, 92–94  
     block diagrams in, 40  
     categories of, 266  
     code in, 41  
     compiling, 41  
     configuration, 268–269  
     data space in, 40–41  
     debugging of, 54  
     defining, 5  
     description documentation of, 231–232, 239, 240

executing of, 54  
     front panel of, 40  
     hierarchy of, 201–202  
     organization of, 391–393  
     parts of, 40–41  
     of plug-and-play instrument drivers, 266  
     plug-in, 425–429  
     real instrumentation vs., 4–6  
     revision history of, 234–235  
     in RT software design, 391–393

(*See also specific types, e.g.: Autonomous VIs*)

Virtual memory (VM), 547

VIs (*see* Virtual instruments)

VISA API, 245

VISA Configure Serial Port function, 255

## U

UART, 16550-compatible, 246  
 UDP (Universal Datagram Protocol), 482  
 Unbundle, 123  
 Unbundle By Name, 123, 468, 475  
 Underflow, 104, 310  
 Undersampling, 306  
 Undo, 27  
 Unflatten From String, 133  
 Uniform Mechanical Code, 443  
 Uninitialized shift registers, 69–71  
 Unipolar inputs, 315  
 Unique naming, 71  
 Units, 467  
 Universal Datagram Protocol (UDP), 482  
 University of Texas at Austin (UT), 10  
 UNIX systems, 12, 23  
 Unprintables, 110, 111  
 Unsigned integers, 104  
 Up/down pins, 523  
 Upper Limit frame, 488  
 USB devices, 245  
 User interface prototyping, 192–196

User needs analysis, 186–187

User training, 217

User-defined codes, 265

User-defined descriptions, 265

User-defined events, 82, 212

Using LabVIEW to Create

*Multithreaded VIs for Maximum Performance and Reliability (see Application Note 114 (Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability))*

Using Quadrature Encoders with E Series DAQ Boards (AN084), 523

UT (University of Texas at Austin), 10

Utilities, for datalog files, 183

Utility VIs, 271–272

VISA Open, 267  
 VISA Property node, 267  
 VISA serial port functions, 246  
 VISA session identifier, 257, 258  
 VISA Transition Library (VTL), 257  
 Visible Items, 127  
 Vision (IMAQ), 574  
 Vision Builder (IMAQ), 577  
 Vision package (IMAQ), 577  
*The Visual Display of Quantitative Information* (Edward Tufte), 556  
 VM (virtual memory), 547  
 VME bus, 519  
 VMEbus Extensions for Instrumentation (VXI), 6, 311, 519  
 Voltage:  
     analog, 278  
     GPIB and, 278  
     peak, 240  
     waveform data of, 278  
 Volts (Potential) controls, 78  
 VTL (VISA Transition Library), 257  
 VXI (*see* VMEbus Extensions for Instrumentation)  
 VXI instrument, 6

**W**

Wait, abortable, 156  
 Wait (ms), 138, 140

Wait At Rendezvous VI, 165  
 Wait For RQS, 270  
 Wait on Notification, 158–159  
 Wait On Occurrence, 155  
 Wait Until Next ms Multiple, 138, 139  
 Warnings (*see* Alarms)  
 Watchdog program, 458  
 Watcom C, 24  
 Waterfall plots, 563, 565  
 Watts (Power) controls, 78  
 Waveform Chart, 499  
 Waveform Generation VIs, 125  
 Waveform Graph, 558  
 Waveform Measurement utilities, 125  
 Waveform Scan VI, 499  
 Waveforms, 125–127, 558  
     attributes of, 126  
     generation of, 403, 529  
     graphs using, 558–563  
     nanosecond, 542  
     recording of, 516  
     of voltage, 278  
 Wells, George, 291  
 Which Button, 467, 470, 481  
 While Loops, 64–66  
 Wind up, 445  
 WindClose (IMAQ), 579  
 WindDraw (IMAQ), 579  
 WindGetROI (IMAQ), 579  
 Windoid, 55

Windows (OS), 454  
 WindToolsSelect (IMAQ), 579  
 Wire-frame plots, 570  
 Wiring, 180  
 Wiring issues:  
     of GPIB instruments, 253–254  
     of serial instruments,  
         252–253  
 Write Datalog, 181  
 Write File (IMAQ), 579  
 Write function (DataSocket), 483  
 Write Mark, 171  
 Write To Measurement File VI, 176  
 Write to Spreadsheet File, 172  
 Write to Text File, 60, 171  
 Writing:  
     to binary files, 178–179  
     to datalog files, 181–182  
     to text files, 170–172  
 Wrong numbering, 96

**X**

XY Graph, 500, 528, 560

**Z**

Z index pulse, 523  
 Zero (null) terminators, 134  
 Z-scale, 571