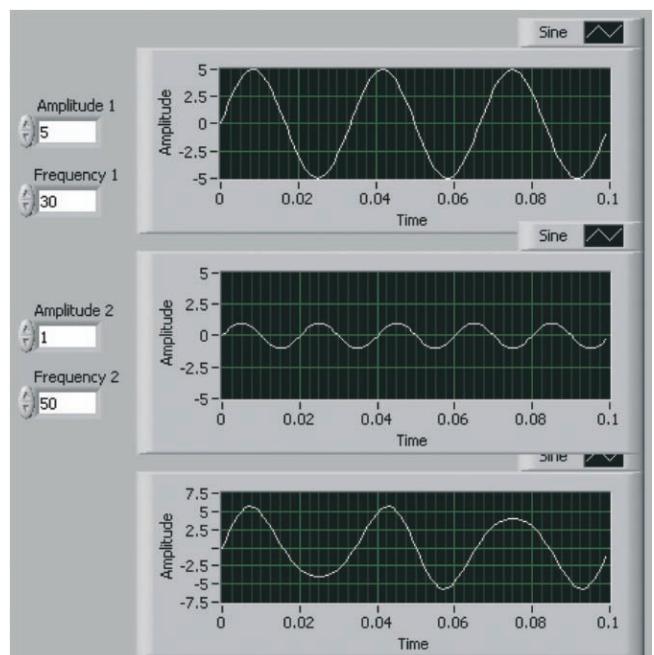
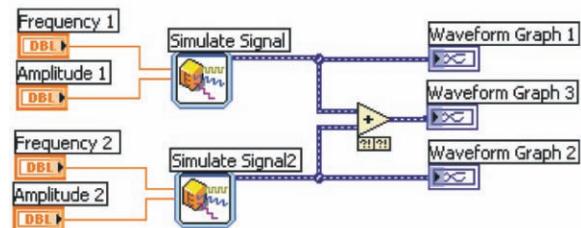
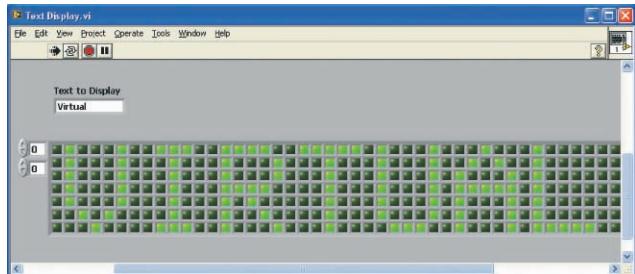
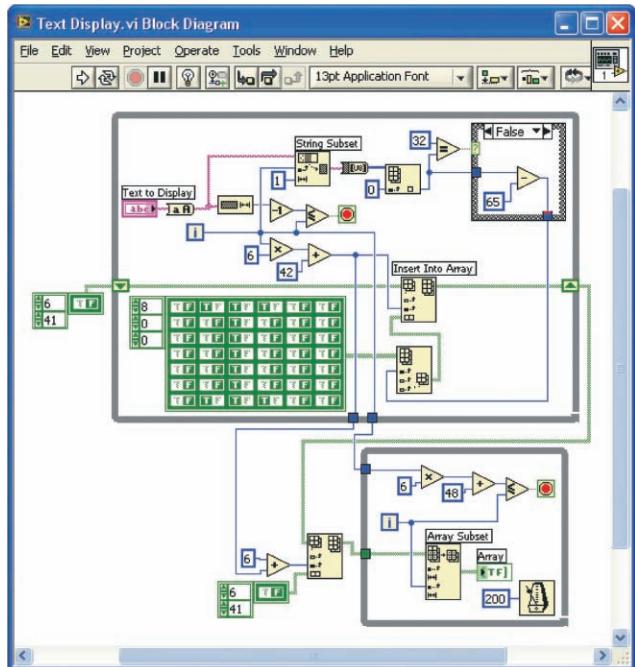


# Virtual Instrumentation Using LabVIEW



**Jovitha Jerome**



# **VIRTUAL INSTRUMENTATION USING LabVIEW**

**Jovitha Jerome**

Professor and Head

Department of Instrumentation and Control Systems Engineering  
PSG College of Technology  
Coimbatore, Tamil Nadu

**PHI Learning Private Limited**

New Delhi-110001  
2010

**Rs. 375.00**

**VIRTUAL INSTRUMENTATION USING LabVIEW**  
Jovitha Jerome

© 2010 by PHI Learning Private Limited, New Delhi. All rights reserved. No part of this book may be reproduced in any form, by mimeograph or any other means, without permission in writing from the publisher.

**Warning and Disclaimer**

While every precaution has been taken in the preparation of this book, the author and the publisher do not guarantee the accuracy, adequacy, or completeness of any information contained in this book. Neither is any liability assumed by the author and the publisher for any damages or loss to your data or your equipment resulting directly or indirectly from the use of the information or instructions contained herein.

**Trademark Acknowledgements**

LabVIEW™ is the registered trademark of National Instruments™. Use of any product or service name in this book should not be regarded as affecting the validity of any trademark or service mark.

**ISBN-978-81-203-4030-5**

The export rights of this book are vested solely with the publisher.

Published by Asoke K. Ghosh, PHI Learning Private Limited, M-97, Connaught Circus, New Delhi-110001 and Printed by Mudrak, 30-A, Patparganj, Delhi-110091.

*To*  
My Husband, Daughter and Son

**M. Jerome Benjamin  
Arumika Jerome  
Jude Prabu Jerome**

# CONTENTS

---

<i>Foreword</i> .....	<i>xix</i>
<i>Preface</i> .....	<i>xxi</i>
<b>1 GRAPHICAL SYSTEM DESIGN .....</b>	<b>1–19</b>
1.1 INTRODUCTION .....	1
1.2 GRAPHICAL SYSTEM DESIGN (GSD) MODEL .....	2
1.2.1 Design (Model).....	2
1.2.2 Prototype (Lab).....	3
1.2.3 Deployment (Field).....	4
1.3 DESIGN FLOW WITH GSD .....	5
1.4 VIRTUAL INSTRUMENTATION .....	6
1.5 VIRTUAL INSTRUMENT AND TRADITIONAL INSTRUMENT .....	7
1.6 HARDWARE AND SOFTWARE IN VIRTUAL INSTRUMENTATION .....	10
1.6.1 Role of Hardware in Virtual Instrumentation .....	10
1.6.2 Role of Software in Virtual Instrumentation.....	10
1.7 VIRTUAL INSTRUMENTATION FOR TEST, CONTROL AND DESIGN .....	12
1.7.1 Virtual Instrumentation for Test.....	12
1.7.2 Virtual Instrumentation for Industrial I/O and Control .....	13
1.7.3 Virtual Instrumentation for Design .....	13
1.8 VIRTUAL INSTRUMENTATION IN THE ENGINEERING PROCESS .....	14
1.8.1 Research and Development.....	14
1.8.2 Development Test and Validation .....	14
1.8.3 Manufacturing Test .....	14

1.9	VIRTUAL INSTRUMENTS BEYOND PERSONAL COMPUTER .....	15
1.10	GRAPHICAL SYSTEM DESIGN USING LabVIEW .....	16
1.11	GRAPHICAL PROGRAMMING AND TEXTUAL PROGRAMMING .....	17
	<i>SUMMARY</i> .....	18
	<i>REVIEW QUESTIONS</i> .....	19
<b>2</b>	<b>INTRODUCTION TO LabVIEW .....</b>	<b>20–46</b>
2.1	INTRODUCTION .....	20
2.2	ADVANTAGES OF LabVIEW .....	21
2.3	SOFTWARE ENVIRONMENT .....	23
2.3.1	Front Panel Windows .....	23
2.3.2	Block Diagram Windows .....	24
2.3.3	Icon/Connector Pane .....	25
2.4	CREATING AND SAVING A VI .....	25
2.5	FRONT PANEL TOOLBAR .....	26
2.6	BLOCK DIAGRAM TOOLBAR .....	27
2.7	PALETTES .....	28
2.7.1	Tools Palette .....	28
2.7.2	Front Panel—Controls Palette .....	29
2.7.3	Block Diagram—Functions Palette .....	30
2.8	SHORTCUT MENUS .....	31
2.9	PROPERTY DIALOG BOXES .....	32
2.10	FRONT PANEL CONTROLS AND INDICATORS .....	33
2.11	BLOCK DIAGRAM .....	34
2.11.1	Terminals .....	34
2.11.2	Nodes .....	35
2.11.3	Functions .....	35
2.11.4	SubVIs .....	35
2.11.5	Express VIs and VIs .....	35
2.11.6	Wires .....	35
2.12	DATA TYPES .....	36
2.13	DATA FLOW PROGRAM .....	36
2.14	LabVIEW DOCUMENTATION RESOURCES .....	37
2.15	KEYBOARD SHORTCUTS .....	38
	<i>SUMMARY</i> .....	38
	<i>MISCELLANEOUS SOLVED PROBLEMS</i> .....	39
	<i>REVIEW QUESTIONS</i> .....	45
	<i>EXERCISES</i> .....	46

---

<b>3 MODULAR PROGRAMMING .....</b>	<b>47–64</b>
3.1 INTRODUCTION .....	47
3.2 MODULAR PROGRAMMING IN LabVIEW .....	48
3.3 BUILD A VI FRONT PANEL AND BLOCK DIAGRAM .....	49
3.4 ICON AND CONNECTOR PANE .....	49
3.5 CREATING AN ICON .....	50
3.6 BUILDING A CONNECTOR PANE .....	51
3.6.1 Assigning Terminals to Controls and Indicators .....	52
3.6.2 Confirming Terminal Connections .....	53
3.6.3 Deleting Terminal Connections .....	53
3.6.4 Setting Required, Recommended, and Optional Inputs and Outputs .....	54
3.7 DISPLAYING SUBVIs AND EXPRESS VIs AS ICONS OR EXPANDABLE NODES .....	54
3.8 CREATING SUBVIs FROM SECTIONS OF A VI.....	55
3.9 OPENING AND EDITING SUBVIs .....	57
3.10 PLACING SUBVIs ON BLOCK DIAGRAMS .....	57
3.11 SAVING SUBVIs .....	57
3.12 CREATING A STAND-ALONE APPLICATION .....	57
3.12.1 Building the Application and Installer .....	58
3.12.2 Application (EXE) Build Specification .....	58
SUMMARY.....	58
MISCELLANEOUS SOLVED PROBLEMS .....	59
REVIEW QUESTIONS .....	64
EXERCISES .....	64
<b>4 REPETITION AND LOOPS .....</b>	<b>65–90</b>
4.1 INTRODUCTION .....	65
4.2 FOR LOOPS .....	65
4.3 WHILE LOOPS .....	67
4.4 STRUCTURE TUNNELS .....	70
4.5 TERMINALS INSIDE OR OUTSIDE LOOPS .....	71
4.6 SHIFT REGISTERS .....	71
4.6.1 Initializing Shift Registers .....	73
4.6.2 Stacked Shift Registers .....	74
4.6.3 Replacing Tunnels with Shift Registers .....	74
4.6.4 Replacing Shift Registers with Tunnels .....	75
4.7 FEEDBACK NODES .....	75
4.7.1 Initializing a Feedback Node .....	76
4.8 CONTROL TIMING .....	77

4.9	COMMUNICATING AMONG MULTIPLE LOOPS .....	79
4.10	LOCAL VARIABLES .....	79
4.11	GLOBAL VARIABLES .....	81
	SUMMARY.....	83
	MISCELLANEOUS SOLVED PROBLEMS .....	83
	REVIEW QUESTIONS .....	89
	EXERCISES .....	89
<b>5</b>	<b>ARRAYS .....</b>	<b>91–114</b>
5.1	INTRODUCTION .....	91
5.2	ARRAYS IN LabVIEW .....	91
5.3	CREATING ONE-DIMENSIONAL ARRAY CONTROLS, INDICATORS AND CONSTANTS .....	92
5.4	CREATING TWO-DIMENSIONAL ARRAYS .....	93
5.5	CREATING MULTIDIMENSIONAL ARRAYS .....	94
5.6	INITIALIZING ARRAYS .....	94
5.7	DELETING ELEMENTS, ROWS, COLUMNS AND PAGES WITHIN ARRAYS... 95	
5.8	INSERTING ELEMENTS, ROWS, COLUMNS AND PAGES INTO ARRAYS .... 96	
5.9	REPLACING ELEMENTS, ROWS, COLUMNS, AND PAGES WITHIN ARRAYS .....	97
5.10	ARRAY FUNCTIONS .....	98
5.11	AUTO INDEXING .....	100
5.12	CREATING TWO-DIMENSIONAL ARRAYS USING LOOPS .....	100
5.13	IDENTIFICATION OF DATA STRUCTURE (SCALAR AND ARRAY) USING WIRES .....	101
5.14	USING AUTO-INDEXING TO SET THE FOR LOOP COUNT .....	101
5.15	MATRIX OPERATIONS WITH ARRAYS .....	102
	5.15.1 Converting an Array to a Matrix .....	103
	5.15.2 Converting a Matrix to an Array .....	103
5.16	POLYMORPHISM .....	104
	SUMMARY.....	106
	MISCELLANEOUS SOLVED PROBLEMS .....	106
	REVIEW QUESTIONS .....	113
	EXERCISES .....	113
<b>6</b>	<b>CLUSTERS .....</b>	<b>115–130</b>
6.1	INTRODUCTION .....	115
6.2	CREATING CLUSTER CONTROLS AND INDICATORS .....	116
6.3	CREATING CLUSTER CONSTANT .....	117

---

6.4	ORDER OF CLUSTER ELEMENTS .....	117
6.5	CLUSTER OPERATIONS .....	118
6.6	ASSEMBLING CLUSTERS .....	118
6.7	DISASSEMBLING CLUSTERS .....	120
6.8	CONVERSION BETWEEN ARRAYS AND CLUSTERS .....	121
6.9	ERROR HANDLING.....	122
6.10	ERROR CLUSTER .....	123
	<i>SUMMARY.....</i>	124
	<i>MISCELLANEOUS SOLVED PROBLEMS .....</i>	124
	<i>REVIEW QUESTIONS .....</i>	129
	<i>EXERCISES .....</i>	130
<b>7</b>	<b>PLOTTING DATA .....</b>	<b>131–159</b>
7.1	INTRODUCTION .....	131
7.2	TYPES OF WAVEFORMS .....	131
7.3	WAVEFORM GRAPHS .....	132
7.3.1	Displaying a Single Plot on Waveform Graphs .....	132
7.3.2	Displaying Multiple Plots on Waveform Graphs .....	132
7.4	WAVEFORM CHARTS .....	133
7.4.1	Displaying a Single Plot on Waveform Charts .....	134
7.4.2	Displaying Multiple Plots on Waveform Charts .....	134
7.5	WAVEFORM DATA TYPE .....	135
7.6	XY GRAPHS .....	135
7.6.1	Displaying a Single Plot on XY Graphs .....	136
7.6.2	Displaying Multiple Plots on XY Graphs .....	136
7.7	INTENSITY GRAPHS AND CHARTS .....	136
7.7.1	Intensity Charts .....	137
7.7.2	Intensity Graphs .....	138
7.7.3	Using Color Mapping with Intensity Graphs and Charts .....	138
7.8	DIGITAL WAVEFORM GRAPHS .....	139
7.8.1	Digital Waveform Data Type .....	140
7.9	3D GRAPHS .....	141
7.10	CUSTOMIZING GRAPHS AND CHARTS .....	142
7.10.1	Using Multiple X- and Y-Scales .....	142
7.10.2	Autoscaling .....	142
7.10.3	Formatting X- and Y-Scales .....	142
7.10.4	Using the Graph Palette .....	143
7.10.5	Customizing Graph and Chart Appearance .....	143
7.10.6	Exporting Images of Graphs, Charts, and Tables .....	143

7.11	CUSTOMIZING GRAPHS .....	144
7.11.1	Using Graph Cursors .....	144
7.11.2	Using Graph Annotations .....	145
7.12	CUSTOMIZING 3D GRAPHS .....	146
7.13	CUSTOMIZING CHARTS .....	146
7.13.1	Configuring Chart History Length.....	146
7.13.2	Configuring Chart Update Modes .....	147
7.13.3	Using Overlaid and Stacked Plots .....	147
7.14	DYNAMICALLY FORMATTING WAVEFORM GRAPHS .....	148
7.15	CONFIGURING A GRAPH OR CHART .....	148
7.16	DISPLAYING SPECIAL PLANES ON THE XY GRAPH.....	149
7.16.1	Displaying a Nyquist Plane .....	149
7.16.2	Displaying a Nichols Plane .....	150
7.16.3	Displaying an S Plane .....	150
7.16.4	Displaying a Z Plane .....	151
SUMMARY.....		152
<i>MISCELLANEOUS SOLVED PROBLEMS</i> .....		153
<i>REVIEW QUESTIONS</i> .....		159
<i>EXERCISES</i> .....		159
<b>8</b>	<b>STRUCTURES .....</b>	<b>160–193</b>
8.1	INTRODUCTION .....	160
8.2	CASE STRUCTURES .....	161
8.2.1	Case Selector Values and Data Types .....	162
8.2.2	Input and Output Tunnels .....	162
8.2.3	Using Case Structures for Error Handling .....	163
8.3	SEQUENCE STRUCTURES .....	163
8.3.1	Flat Sequence Structure .....	163
8.3.2	Stacked Sequence Structure .....	163
8.3.3	Using Sequence Structures .....	164
8.3.4	Avoiding Overuse Sequence Structures .....	164
8.3.5	Adding and Removing Sequence Local Terminals .....	165
8.4	CUSTOMIZING STRUCTURES .....	166
8.4.1	Placing Structures on the Block Diagram.....	166
8.4.2	Placing Objects inside Structures .....	166
8.4.3	Removing Structures without Deleting Objects in the Structure .....	167
8.4.4	Resizing Structures .....	167
8.4.5	Adding Cases to the Middle of an Ordered List .....	167
8.4.6	Adding, Duplicating and Deleting Subdiagrams .....	168

---

8.5	TIMED STRUCTURES .....	168
8.5.1	Timed Loop Structure .....	169
8.5.2	Timed Sequence Structure .....	170
8.5.3	Timed Loop with Frames Structure.....	170
8.6	FORMULA NODES .....	171
8.6.1	Using the Formula Node.....	172
8.6.2	Creating Formula Nodes .....	173
8.6.3	Formula Node Syntax .....	174
8.6.4	Scope Rules for Declaring Variables in Formula Nodes.....	174
8.7	EVENT STRUCTURE .....	175
8.8	LabVIEW MathScript .....	176
8.8.1	LabVIEW MathScript Window .....	176
8.8.2	MathScript Node .....	177
8.8.3	Creating a LabVIEW MathScript .....	177
8.8.4	Defining a MathScript Function or Script .....	178
8.8.5	Debugging Scripts .....	178
8.8.6	Clearing, Saving and Loading Scripts .....	178
8.8.7	Importing or Exporting Scripts .....	179
8.8.8	Configuring the Data Type of Script Node Terminals .....	180
8.8.9	MathScript Function Syntax .....	180
8.8.10	MathScript Syntax .....	181
	<i>SUMMARY</i> .....	181
	<i>MISCELLANEOUS SOLVED PROBLEMS</i> .....	182
	<i>REVIEW QUESTIONS</i> .....	193
	<i>EXERCISES</i> .....	193
<b>9</b>	<b>STRINGS AND FILE I/O .....</b>	<b>194–222</b>
9.1	INTRODUCTION .....	194
9.2	CREATING STRING CONTROLS AND INDICATORS .....	194
9.2.1	Tables .....	195
9.3	STRING FUNCTIONS .....	196
9.3.1	Converting Numeric Values to Strings with the Build Text Express VI .....	198
9.3.2	Converting Strings to Numeric Values with the Scan From String Function .....	199
9.4	EDITING, FORMATTING AND PARSING STRINGS .....	199
9.5	FORMATTING STRINGS .....	200
9.5.1	Array to Spreadsheet String .....	200
9.5.2	Spreadsheet String to Array .....	200
9.5.3	Scan From String .....	201

9.5.4	Format into String .....	201
9.5.5	Format Value .....	201
9.5.6	Scan Value .....	201
9.5.7	Scan from File .....	202
9.5.8	Format into File .....	202
9.6	CONFIGURING STRING CONTROLS AND INDICATORS .....	202
9.6.1	Normal Display .....	203
9.6.2	Backslash ('\\') Codes Display .....	203
9.6.3	Password Display .....	203
9.6.4	Hex Display .....	203
9.6.5	Limit to Single Line .....	203
9.6.6	Update Value while Typing .....	203
9.6.7	Enable Wrapping .....	203
9.7	BASICS OF FILE INPUT/OUTPUT .....	204
9.8	CHOOSING A FILE I/O FORMAT .....	204
9.8.1	Use of Text Files .....	205
9.8.2	Use of Binary Files .....	205
9.8.3	Use of Datalog Files .....	205
9.9	LabVIEW DATA DIRECTORY .....	206
9.10	FILE I/O VIs .....	206
9.10.1	Disk Streaming with Low-Level Functions .....	207
9.10.2	High Level File I/O .....	207
9.11	CREATING A RELATIVE PATH .....	208
	SUMMARY .....	208
	MISCELLANEOUS SOLVED PROBLEMS .....	209
	REVIEW QUESTIONS .....	221
	EXERCISES .....	221
<b>10</b>	<b>INSTRUMENT CONTROL .....</b>	<b>223–252</b>
10.1	INTRODUCTION .....	223
10.2	GPIB COMMUNICATION .....	224
10.2.1	Data Transfer Termination .....	225
10.2.2	Data Transfer Rate .....	226
10.3	HARDWARE SPECIFICATIONS .....	226
10.4	SOFTWARE ARCHITECTURE .....	227
10.4.1	MAX (Windows; GPIB) .....	228
10.5	INSTRUMENT I/O ASSISTANT .....	228
10.6	VISA .....	230
10.6.1	VISA Programming Terminology .....	231
10.6.2	VISA and Serial .....	232

---

10.7	INSTRUMENT DRIVERS .....	232
10.7.1	Instrument Driver VIs .....	234
10.8	SERIAL PORT COMMUNICATIONS .....	235
10.8.1	Data Transfer Rate .....	237
10.8.2	Serial Port Standards .....	238
10.9	USING OTHER INTERFACES .....	239
	<i>SUMMARY</i> .....	239
	<i>MISCELLANEOUS SOLVED PROBLEMS</i> .....	239
	<i>REVIEW QUESTIONS</i> .....	251
	<i>EXERCISE</i> .....	252
<b>11</b>	<b>DATA ACQUISITION .....</b>	<b>253–292</b>
11.1	INTRODUCTION .....	253
11.2	TRANSDUSERS .....	254
11.3	SIGNALS .....	254
11.3.1	Analog Signals.....	255
11.3.2	Digital Signals .....	256
11.4	SIGNAL CONDITIONING .....	256
11.4.1	Amplification .....	258
11.4.2	Isolation .....	258
11.4.3	Multiplexing .....	258
11.4.4	Filtering .....	259
11.4.5	Transduces Excitation.....	259
11.4.6	Linearization .....	259
11.5	DAQ HARDWARE CONFIGURATION .....	260
11.5.1	Measurement & Automation Explorer.....	260
11.5.2	Scales .....	260
11.5.3	Simulating a DAQ Device .....	260
11.6	DAQ HARDWARE .....	261
11.6.1	Terminal Block and Cable .....	262
11.6.2	DAQ Signal Accessory .....	262
11.6.3	DAQ Device .....	263
11.7	ANALOG INPUTS .....	265
11.7.1	Analog-to-Digital Conversion .....	265
11.7.2	Task Timing .....	265
11.7.3	Task Triggering .....	266
11.8	ANALOG OUTPUTS .....	266
11.8.1	Task Timing .....	266
11.8.2	Task Triggering .....	267
11.8.3	Digital-to-Analog Conversion .....	267

11.9 COUNTERS .....	267
11.10 Digital I/O (DIO) .....	268
11.11 DAQ Software Architecture .....	269
11.11.1 Driver Software .....	269
11.11.2 Application Software .....	270
11.12 DAQ ASSISTANT .....	270
11.12.1 Launch the DAQ Assistant .....	271
11.12.2 Create the Task .....	272
11.12.3 Configure the Task .....	272
11.12.4 Test the Task .....	273
11.12.5 Generate LabVIEW Code .....	273
11.13 CHANNELS AND TASK CONFIGURATION .....	275
11.14 SELECTING AND CONFIGURING A DATA ACQUISITION DEVICE .....	276
11.14.1 Signal Sources .....	276
11.14.2 Measurement Systems .....	277
11.14.3 Increasing Measurement Quality .....	279
11.14.4 Increasing Shape Recovery .....	281
11.15 COMPONENTS OF COMPUTER BASED MEASUREMENT SYSTEM .....	282
11.15.1 Installing the DAQ Card .....	284
SUMMARY .....	287
MISCELLANEOUS SOLVED PROBLEMS .....	288
REVIEW QUESTIONS .....	291
EXERCISES .....	292
<b>12 IMAQ VISION .....</b>	<b>293–324</b>
12.1 VISION BASICS .....	293
12.1.1 Digital Images .....	293
12.1.2 Display .....	295
12.1.3 System Setup and Calibration .....	295
12.2 IMAGE PROCESSING AND ANALYSIS .....	296
12.2.1 Image Analysis .....	296
12.2.2 Image Processing .....	297
12.2.3 Operators .....	298
12.2.4 Frequency Domain Analysis .....	298
12.3 PARTICLE ANALYSIS .....	299
12.3.1 Thresholding .....	299
12.3.2 Binary Morphology .....	300
12.3.3 Particle Measurements .....	300

---

12.4	MACHINE VISION .....	301
12.4.1	Edge Detection .....	302
12.4.2	Pattern Matching .....	303
12.4.3	Geometric Matching .....	303
12.4.4	Dimensional Measurements .....	304
12.4.5	Color Inspection .....	305
12.4.6	Binary Partical Classification .....	307
12.4.7	Optical Character Recognition .....	308
12.4.8	Instrument Readers .....	308
12.5	MACHINE VISION HARDWARE AND SOFTWARE .....	309
12.6	BUILDING A COMPLETE MACHINE VISION SYSTEM .....	310
12.7	ACQUIRING AND DISPLAYING IMAGES WITH NI-IMAQ DRIVER SOFTWARE .....	311
12.8	IMAGE PROCESSING TOOLS AND FUNCTIONS IN IMAQ VISION .....	312
13.9	MACHINE VISION APPLICATION AREAS .....	317
	SUMMARY.....	318
	<i>MISCELLANEOUS SOLVED PROBLEMS</i> .....	319
	<i>REVIEW QUESTIONS</i> .....	323
	<i>EXERCISES</i> .....	323
<b>13</b>	<b>MOTION CONTROL .....</b>	<b>325–361</b>
13.1	COMPONENTS OF A MOTION CONTROL SYSTEM .....	325
13.2	SOFTWARE FOR CONFIGURATION, PROTOTYPING AND DEVELOPMENT .....	326
13.2.1	Configuration .....	326
13.2.2	Prototyping .....	328
13.2.3	Development .....	329
13.3	MOTION CONTROLLER .....	330
13.3.1	Calculating the Trajectory .....	330
13.3.2	Selecting the Right Motion Controller .....	331
13.3.3	Creating Custom Motion Controllers .....	331
13.4	MOVE TYPES .....	331
13.4.1	Single-Axis, Point-to-Point Motion .....	331
13.4.2	Coordinated Multi-Axis Motion .....	332
13.4.3	Blended Motion .....	332
13.4.4	Contoured Motion .....	333
13.4.5	Electronic Gearing .....	334
13.5	MOTOR AMPLIFIERS AND DRIVES .....	334
13.5.1	Simple Servo Amplifiers .....	335
13.5.2	Stepper Motor Amplifiers .....	335

13.5.3	AC Servo Amplifiers .....	336
13.5.4	DC Servo Amplifiers .....	336
13.6	MOTOR FUNDAMENTALS .....	336
13.6.1	Servomotors .....	337
13.6.2	Brushless Servomotors .....	338
13.6.3	Servomotor Applications .....	339
13.6.4	Stepper Motors .....	344
13.6.5	Stepper Motor Types .....	344
13.6.6	Linear Stepper Motors .....	347
13.6.7	Stepper Motor Applications .....	350
13.7	FEEDBACK DEVICES AND MOTION I/O .....	351
13.7.1	Encoders .....	352
13.7.2	Linear and Rotary Encoders .....	353
13.7.3	Resolvers .....	355
13.7.4	Tachometers .....	356
13.7.5	Magnetic Encoders .....	357
13.7.6	Optical Encoders .....	358
13.7.7	Quadrature Encoders .....	358
13.8	MOTION I/O .....	358
13.8.1	Integration with Motion using RTSI .....	359
	SUMMARY.....	360
	REVIEW QUESTIONS .....	361
<b>14</b>	<b>LabVIEW TOOL AND GSD APPLICATIONS .....</b>	<b>362–387</b>
14.1	INTRODUCTION .....	362
14.2	SIGNAL PROCESSING AND ANALYSIS .....	362
14.3	PROFESSIONAL DEVELOPMENT TOOLS .....	363
14.4	THIRD-PARTY CONNECTIVITY TOOLS .....	364
14.5	CONTROL DESIGN AND SIMULATION TOOLS .....	364
14.6	DIGITAL FILTER DESIGN TOOLKIT .....	365
14.7	SOUND AND VIBRATION TOOLKIT .....	367
14.8	MODULATION TOOLKIT .....	368
14.9	SPECTRAL MEASUREMENTS TOOLKIT .....	369
14.10	EXPRESS VI DEVELOPMENT TOOLKIT .....	371
14.11	REPORT GENERATION TOOLKIT FOR MICROSOFT OFFICE .....	373
14.12	SIMULATION INTERFACE TOOLKIT .....	374
14.13	CONTROL DESIGN AND SIMULATION MODULE .....	375
14.14	CONTROL DESIGN TOOLKIT .....	375
14.15	PID CONTROL TOOLKIT .....	377
14.16	SIMULATION MODULE .....	378

---

14.17 SYSTEM IDENTIFICATION TOOLKIT .....	379
14.18 DIAdem .....	379
14.19 DATA LOGGING AND SUPERVISORY CONTROL .....	380
14.20 EMBEDDED MODULE .....	381
14.21 BIOMEDICAL STARTUP KIT .....	381
14.22 GSD APPLICATIONS .....	382
14.22.1 Material Handling System .....	382
14.22.2 Plastic Injection Molding System .....	384
14.22.3 Semiconductor Production Control System .....	385
SUMMARY.....	386
REVIEW QUESTIONS .....	387
<b>INDEX .....</b>	<b>389–392</b>

# FOREWORD

---

Dr. Jovitha Jerome has been associated with the development of Virtual Instrumentation in India right from early 2003 and it was appropriate that the concepts she had used for teaching and in her lab be compiled into a textbook. I was very happy to write the foreword to the book on my most favourite concept. I also felt that the concept of Virtual Instrumentation is best explained only when it is supported by a good set of examples from a software perspective as done in her book *Virtual Instrumentation Using LabVIEW*. It is evident by going through the chapters in the textbook how several years of work has now evolved into a complete textbook for students to understand and apply Virtual Instrumentation during their engineering coursework.

The book builds the concept of Virtual Instrumentation by using clear and concise flow of programming elements using LabVIEW as the application development environment. It goes beyond the traditional approach to instrumentation vis-à-vis the Virtual Instrumentation to touch upon Graphical System Design and Algorithm Engineering. The textbook also covers the various toolkits available for realizing the applications in different domains of control, communication, image processing, biomedical and signal processing. The perspective of Acquire, Analyze and Present which are the three components of Virtual Instrumentation are well brought across in the book. This enables one to learn the subject of Virtual Instrumentation not just as a programming language but as an application design, prototype and deploy platform. The textbook is designed to suit multi-disciplinary streams of engineering and can be used by non-circuit branches like Mechanical and Civil engineering to teach the concepts of sensors, data acquisition, instrumentation and signal processing.

The textbook deals with each concept of instrumentation with multiple examples, thereby enabling it to be used for both classroom teaching and lab exercises. By also addressing the programming constructs in LabVIEW, the book clearly articulates the emerging importance of software in instrumentation systems and embedded systems.

The textbook also goes beyond the simulation comfort zone and deals with specific data acquisition examples and the tasks which students can accomplish by connecting different sensors

and actuators. This will enable hands-on learning of difficult-to-explain engineering concepts. The book subtly brings out the fact that it is possible to move beyond the realm of mathematical modeling and accomplish simple practical projects in Virtual Instrumentation by connecting to real-world sensors and signals.

I am sure that the textbook will serve as a valuable resource of UG and PG engineering courses for students and faculty using experiential teaching–learning methodology.



**Jayaram Pillai**  
MD, India, Russia and Arabia  
National Instruments

# PREFACE

---

The increased functionality of advanced electronics is possible because devices have become more software centric. Technology never stands still. Engineers and scientists can improve functionality through software instead of developing further specific electronics to do a particular job. The efficient way to meet these demands is to use test and control architectures that are software centric. Graphical system design (GSD) is a modern approach to designing an entire system. Graphical system design competing in today's global economy requires companies to rapidly enter the market with innovative products that offer increased functionality and operate flawlessly. The benefits of graphical system design comprise reduced time to market, optimal system scalability, quick design iteration and increased performance at lower cost.

GSD brings a software platform combined with a hardware platform that offers the opportunity to significantly reduce development cost and time to market. A software platform which integrates multiple models of computation minimizes the time to implement specifications into a design. A flexible commercial-off-the-shelf (COTS) hardware prototyping platform which supports the software platform and offers customizable components minimizes the time to first prototype because the time and money required for designing custom hardware is eliminated. Additionally, prototyping with real input/output results in a higher quality design—eliminating the design failures we see today. Finally, consistent graphical software from the design to prototyping platform and the final deployment target maximizes code reuse and eases the transition to final deployment.

The PSG-NI Virtual Instrumentation Centre was awarded the **Best Graphical System Design Lab** among 240 Virtual Instrumentation centres all over India. The award was initiated by NI Systems (India) Pvt. Ltd., Bangalore (The Indian Branch of National Instruments, USA). The award was based on the student activity/training and the research work carried out by the faculty and was given during the Educators Day 2008 Conference conducted by National Instruments. This book is based on my extensive experience in the Virtual Instrumentation Centre and caters to the needs of practicing technologists and academic community who wish to work on LabVIEW.

The miscellaneous solved problems provided after every chapter aid in comprehension of the principles involved in LabVIEW programming.

Chapter 1 provides an introduction to graphical system design. The graphical system design approach for test, control and embedded design meets this need by providing a unified platform for designing, prototyping and deploying applications. Virtual instrumentation, a comparison of an virtual instrument and a traditional instrument, the hardware and software used to create virtual instruments, and LabVIEW as data flow programming are vividly explained. A brief history of the development of LabVIEW is discussed.

Chapter 2 is on introduction to LabVIEW which extensively explains the graphical programming language. A large number of problems have been solved to help instructors and students easily understand basic LabVIEW programming and create a virtual instrument (VI). It highlights the basic features and techniques used in LabVIEW.

The modular programming concept is explained in Chapter 3. SubVI makes LabVIEW programs easier to read and write, simple to debug, and easy to reuse. Procedures to create a VI as a subVI are clearly explained. First a VI is built and then the icon and the connector pane are created. The power of LabVIEW lies in the hierarchical nature of the VI.

FOR and WHILE loops are discussed in Chapter 4. A FOR loop executes a subdiagram for a set number of times and a WHILE loop executes a subdiagram until a condition is met. The concept of structure tunnels, shift registers, feedback nodes, control timing, local variables and global variables are presented with simple examples.

Arrays group data elements of the same type. Chapter 5 explains how arrays are intrinsic in all programming applications. One-, two- and multidimensional arrays are explained with examples. Array initializing and deleting, auto indexing and polymorphism are also discussed clearly.

Creating cluster controls and indicators are dealt with in Chapter 6. Creating cluster constants, finding the order of cluster elements, cluster functions, error handling and error clusters are all described.

Chapter 7 deals with charts and graphs, several ways to use them, and some of their special features are highlighted. This chapter also describes LabVIEW's special intensity charts and graphs, 3D graphs and digital waveform graph. It also explains waveform data type—a useful LabVIEW representation for time-based data.

Chapter 8 introduces methods for making decisions in a VI. These methods include case structure, sequence structure, event structure, timed structures, diagram disable structure and conditional disable structure. The capabilities of the structures are also described.

Chapter 9 elaborates the use of strings. File input and output is important in most applications. Creating text messages, passing numeric data as character strings to instruments, storing numeric data to disk, instructing or prompting the user with dialog boxes are explained in detail.

The instrument control of stand-alone instruments using a GPIB or serial interface is discussed in Chapter 10. LabVIEW is used to control and acquire data from instruments with the Instrument I/O Assistant, the VISA API and instrument drivers.

Chapter 11 explains the hardware used in a data acquisition system, how to configure the devices and how to program analog input and output, counters, and digital input and output. It provides an overview of each element and explains the most important criteria of these elements. Engineers and scientists can easily acquire, analyze and present data effectively, thus resulting in improved concepts and products.

Chapter 12 on IMAQ vision provides an introduction to vision basics. Image processing and analysis, particle analysis, machine vision, machine vision hardware and software, building a complete machine vision system, acquiring and displaying images with NI-IMAQ driver software, image processing tools and functions in IMAQ vision, and machine vision application areas are some of the interesting topics discussed in this chapter.

The basic components of a motion control system and the software for configuration, prototyping and development are explained in Chapter 13. Motion controller, move types, motor amplifiers and drives, motor fundamentals, feedback devices and motion I/O are elaborately explained. Engineers are empowered to integrate real-world signals sooner for earlier error detection, reuse code for maximum efficiency, benefit immediately from advances in computing technology, and optimize system performance in a way that outpaces traditional design methodologies.

LabVIEW tools and GSD applications are discussed in Chapter 14. An introduction is provided to some important LabVIEW tools like signal processing and analysis tools, control design and simulation tools, sound and vibration toolkit, express VI development toolkit, system identification toolkit, data logging and supervisory control tools and embedded module. A few industrial GSD applications like a material handling system, plastic injection molding system and a semiconductor production control system are discussed.

I thank all my students who suggested that I write this book and all those who encouraged me in this venture. I acknowledge with gratitude Mr. Jayaraman Pillai, Managing Director and Mr. Dhanabal, Academic Manager, National Instruments, Bangalore for all their support and constant encouragement. My whole-hearted gratitude to the Managing Trustee, Chief Executive Officer and the Principal, PSG College of Technology, Coimbatore. I wish to thank all the faculty and staff of the Department of Instrumentation and Control Systems Engineering, PSG College of Technology, Coimbatore and a very special mention to Ms. Vijayalalitha. Finally, I am greatly indebted to my family—my husband Jerome Benjamin, our daughter Arumika Jerome and our son Jude Prabu Jerome and appreciate their encouragement and loving support.

**Jovitha Jerome**



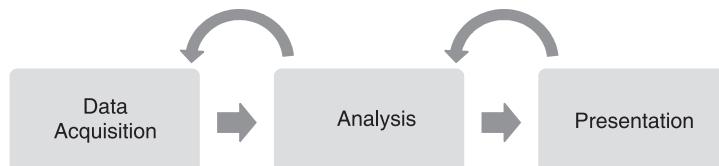
# GRAPHICAL SYSTEM DESIGN

---

## 1.1 INTRODUCTION

The term “scientific computing,” or “computational science”, has been used for many years to define the use of computers (software and hardware) for solving problems related to science and engineering, usually involving experimental or applied research, modeling, and simulation. In a simplistic definition, it refers to the use of computers in solving scientific problems. Scientific computing applications usually follow a three-step process: data acquisition, data analysis and data visualization/presentation.

This three-step approach has been one of the pillars of the NI (National Instruments) virtual instrumentation model as shown in Figure 1.1 since its original conceptualization in the early 1980s, and has been expanded into a more comprehensive model known as *graphical system design* shown in Figure 1.2. In this new model, the focus is to accelerate the research and development cycle, delivering mathematical models to embedded real-time computers faster and easier. This design-flow acceleration is achieved by using NI LabVIEW software and its G programming language as a common system-level design tool for all the different phases in the design-to-deployment flow.



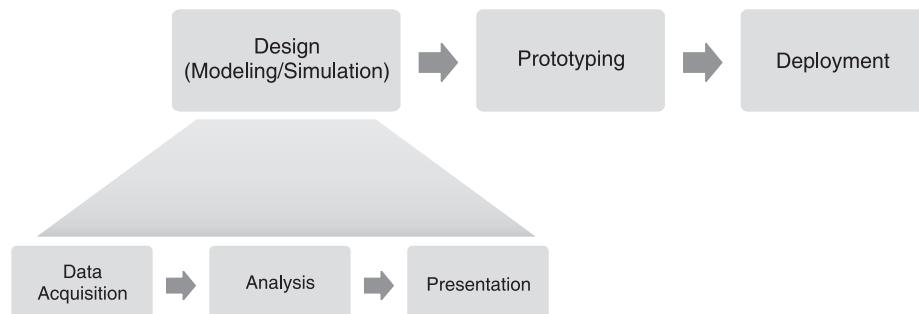
**Figure 1.1** Virtual instrumentation model.



**Figure 1.2** Graphical system design model.

## 1.2 GRAPHICAL SYSTEM DESIGN (GSD) MODEL

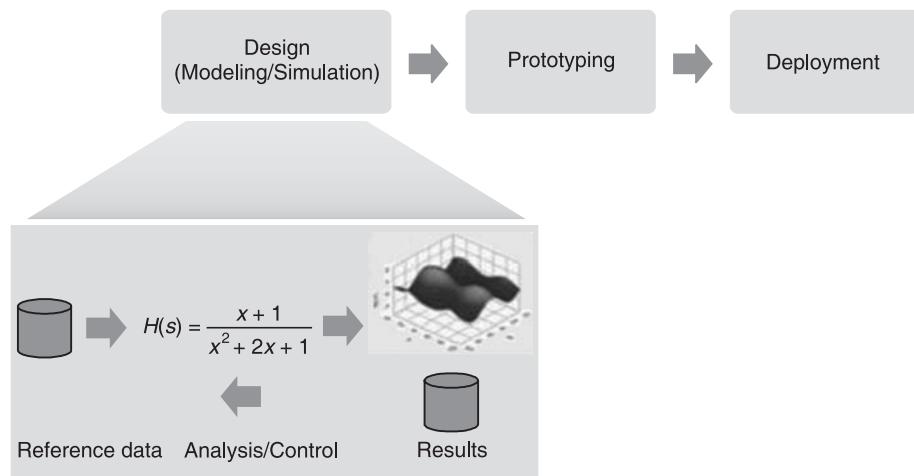
In reality, the virtual instrumentation model is applied in each of the three phases of the graphical system design model as shown in Figure 1.3, because data acquisition, analysis and presentation functions are used in the design, prototyping and deployment phases.



**Figure 1.3** Graphical system design and virtual instrumentation.

### 1.2.1 Design (Model)

In the design phase as shown in Figure 1.4, the researcher develops a mathematical model of the system, including sensors, actuators, plants and controllers, and simulates them under a variety of initial conditions and constraints. The researcher uses different numerical methods with the objective of validating the performance of the model and optimizing it.



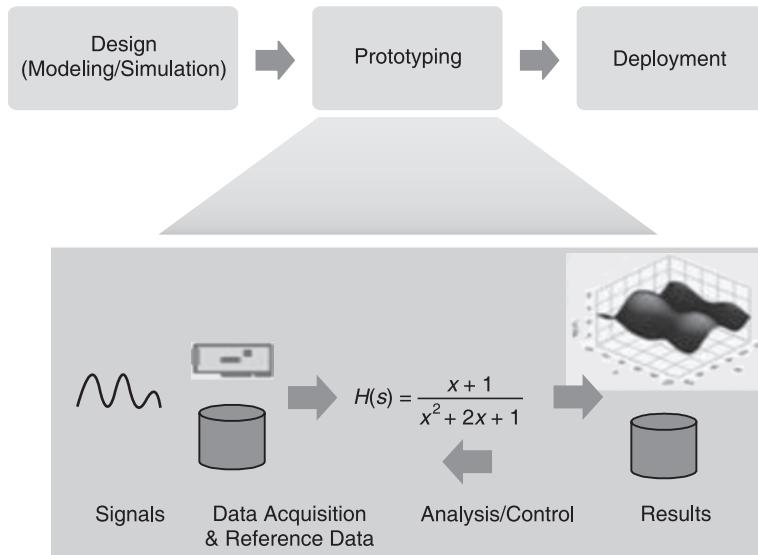
**Figure 1.4** The design phase of the graphical system design model.

In this phase, researchers can acquire reference data from files or databases and incorporate it into the model. A “virtual plant/process” is created, which can be used later for hardware-in-the-loop (HIL) tests. Results from the simulation process are saved for post-analysis and visualization and can be used to introduce changes into the model. This is usually a software-centric process with a strong focus on numerical methods/analysis and mathematics. However, for complex or computationally intensive models, high-performance computing (HPC) using grid computers, standard computers with graphical processing units (GPUs), and multicore based computers is a key factor. In those cases, the hardware has an important impact on the performance of the model solution and simulation. Multicore-ready software tools are scarce, but LabVIEW is multicore-aware. It provides the user with a powerful yet easy-to-use programming language that can take advantage of multicore processors and parallel programming.

### 1.2.2 Prototype (Lab)

If experimental validation of the model is required, researchers develop and test a prototype in the laboratory. Signal processing and analysis as well as visualization can be implemented online while data is being measured and acquired, or while the process is being controlled. The “virtual plant/process” defined in the previous phase can be used for HIL tests. The experimental results obtained in this phase can be used to modify and optimize the original model, which in turn may require additional experiments. Data captured can also be used for system identification and parameter estimation. Usually, this experimental (prototyping) phase is executed on standard PCs or PXI computers, using PCI/PXI data acquisition devices or external measuring devices connected to a PC via USB, Ethernet, GPIB, or serial ports.

This process as shown in Figure 1.5 is usually more software/hardware-centric because sensors, actuators, data acquisition devices, controllers, and the controlled/analyzed plant itself are all key

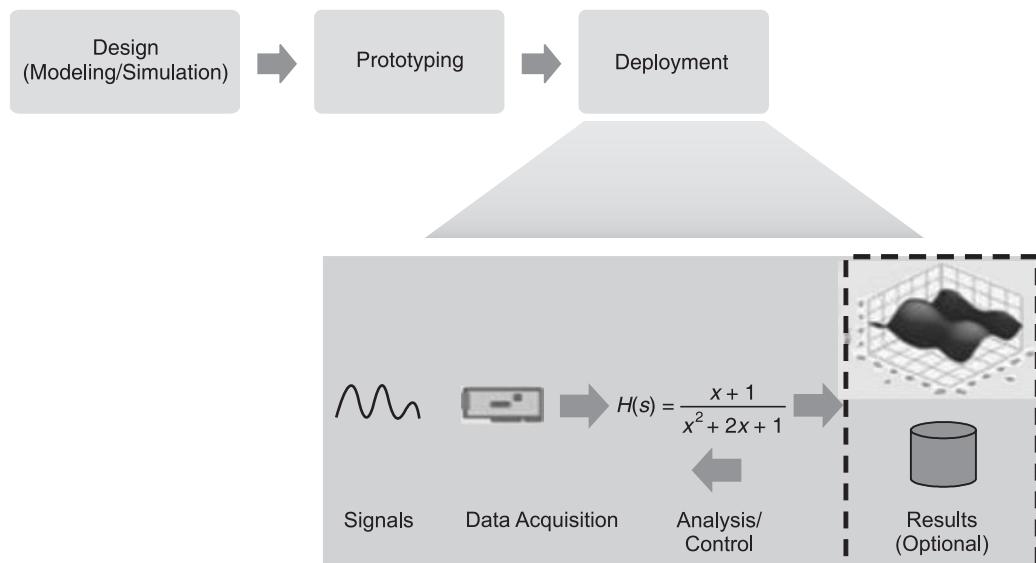


**Figure 1.5** The prototyping phase of the graphical system design model.

elements in the experimental setup. Real-time operating systems (RTOSs) can be used when deterministic performance or higher reliability is required. Also, multicore-based computers can be used when higher computational performance is needed. Field-programmable gate arrays (FPGAs), digital signal processors (DSPs), and GPU processors can be used with the multicore main CPU, allowing LabVIEW to execute critical computations in specialized processors. For example, sets of 2D or 3D differential equations can be solved in real time, or large sets of matrices can be processed at high speed and in real time using parallel programming in multicore computers while sharing part of the signal processing load with an FPGA or a GPU. Data is shared between the multicore processors and an FPGA via DMA first-in-first-out memory buffers (FIFOs). Finally, technical data mining algorithms can be implemented by combining LabVIEW with NI DIAdem, an excellent tool for managing, analyzing, and reporting technical data collected during data acquisition and/or generated during the modelling, simulation or analysis phases.

### 1.2.3 Deployment (Field)

Finally, the model (controller, analyzer or both) is deployed in the field or lab using either a PC (desktop, server or industrial) or PXI, or it can be downloaded to a dedicated embedded controller such as CompactRIO, which usually operates in stand-alone mode and in real-time (deterministic) mode as shown in Figure 1.6. When using the LabVIEW Real-Time Module, symmetric multiprocessing (SMP) techniques can be easily applied. For large systems, with high-channel counts or involving modular instruments such as scopes, digital multimeters (DMMs), RF vector signal analyzers, and dynamic signal acquisition (DSA) devices, the PXI platform is more appropriate. The transition from the prototyping phase to the deployment phase can be very fast and efficient because the same set of tools used for prototyping can, in most cases, be applied to the final deployment of the system in the field.

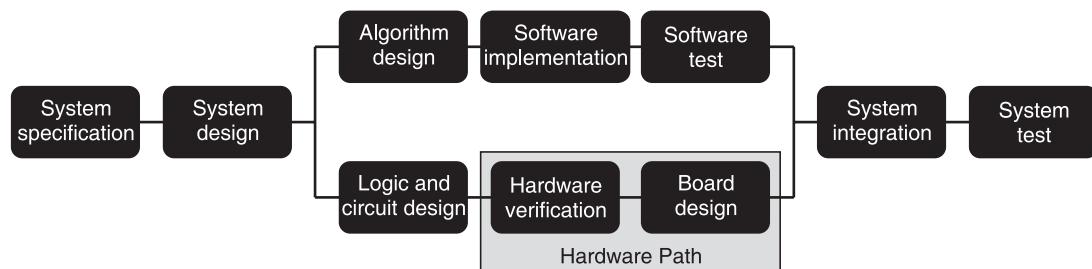


**Figure 1.6** The deployment phase of the graphical system design model.

In many cases, the system is deployed as a headless system (no monitors or interfaces to the user), executing the analysis/control algorithms in real time as a dedicated device. If on-the-field graphical user interfaces (GUIs) or operator interfaces (OIs) are needed, the LabVIEW Touch-Panel Module and industrial-grade monitors with touch-sensitive screens can be used locally. Remotely, data can be shared and visualized via Ethernet with applications developed using LabVIEW and the LabVIEW Datalogging and Supervisory Control Module running on one or more PCs distributed in a network.

### 1.3 DESIGN FLOW WITH GSD

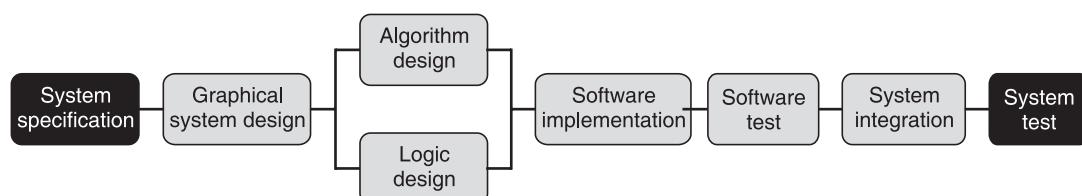
Typical embedded system software and hardware design flows is shown in Figure 1.7.



**Figure 1.7** Typical embedded system software and hardware design flow.

If you are creating custom hardware for final deployment, it is difficult to have the software and hardware developed in parallel as the software is never tested on representative hardware until the process reaches the system integration step. Additionally, you do not want the software development to be purely theoretical, because waiting until the system integration test to include I/O and test the design with real signals may mean that you discover problems too late to meet design deadlines. Most designers currently use a solution like an evaluation board to prototype their systems. However, these boards often only include a few analog and digital I/O channels and rarely include vision, motion, or ability to synchronize I/O. Additionally, designers often have to waste time developing custom boards for sensors or specialized I/O, just to complete a proof of concept.

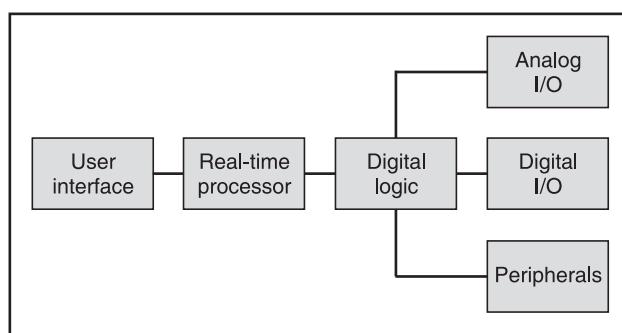
Using flexible Commercial-off-the-shelf (COTS) prototyping platforms instead can truly streamline this process, as shown in Figure 1.8, and eliminates much of the work required for hardware verification and board design. Much like PCs today, in which case anyone can go to an



**Figure 1.8** Stream-lined development flow with graphical system design.

electronics store and plug components such as memory, motherboards and peripheral together to create a PC, graphical system design strives to achieve the same standardization for prototyping platforms. The time has come for a new approach to electronic system design.

For most systems, a prototyping platform must incorporate the same components of the final deployed system. These components are often a real-time processor for executing algorithms deterministically, programmable digital logic for high-speed processing or interfacing the real-time processor to other components, and varied types of I/O and peripherals as shown in Figure 1.9. Finally, as with any system, if the off-the-shelf I/O doesn't serve all of your needs, the platform should also be extensible and customizable when needed.

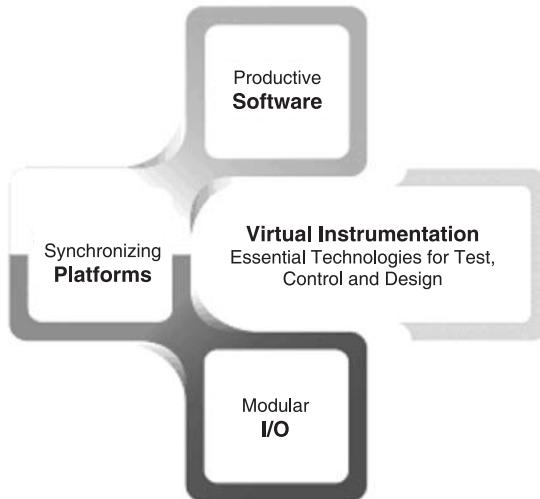


**Figure 1.9** Typical components of an embedded system.

## 1.4 VIRTUAL INSTRUMENTATION

Virtual instrumentation combines mainstream commercial technologies, such as the PC, with flexible software and a wide variety of measurement and control hardware. Engineers use virtual instrumentation to bring the power of flexible software and PC technology to test, control and design applications making accurate analog and digital measurements. Engineers and scientists can create user-defined systems that meet their exact application needs. Industries with automated processes, such as chemical or manufacturing plants use virtual instrumentation with the goal of improving system productivity, reliability, safety, optimization and stability. Virtual instrumentation is computer software that a user would employ to develop a computerized test and measurement system for controlling from a computer desktop, an external measurement hardware device, and for displaying, test or measurement data collected by the external device on instrument-like panels on a computer screen. It extends to computerized systems for controlling processes based on data collected and processed by a computerized instrumentation system. The front panel control function of the existing instrument is duplicated through the computer interface. The application ranges from simple laboratory experiments to large automation application.

Virtual instrumentation as shown in Figure 1.10 uses highly productive software, modular I/O and commercial platforms. National Instruments LabVIEW, a premier virtual instrumentation graphical development environment, uses symbolic or graphical representations to speed up development. The software symbolically represents functions. Consolidating functions within rapidly deployed graphical blocks further speeds up development.



**Figure 1.10** Virtual instrumentation combines productive software, modular I/O and scalable platforms.

Another virtual instrumentation component is modular I/O, designed to be rapidly combined in any order or quantity to ensure that virtual instrumentation can both monitor and control any development aspect. Using well-designed software drivers for modular I/O, engineers and scientists quickly can access functions during concurrent operation.

The third virtual instrumentation element using commercial platforms, often enhanced with accurate synchronization, ensures that virtual instrumentation takes advantage of the very latest computer capabilities and data transfer technologies. This element delivers virtual instrumentation on a long-term technology base that scales with the high investments made in processors, buses and more.

In summary, as innovation mandates use of software to accelerate a new concept and product development, it also requires instrumentation to rapidly adapt to new functionality. Because virtual instrumentation applies software, modular I/O and commercial platforms, it delivers instrumentation capabilities uniquely qualified to keep pace with today's concept and product development.

## 1.5 VIRTUAL INSTRUMENT AND TRADITIONAL INSTRUMENT

A traditional instrument is designed to collect data from an environment, or from a unit under test, and to display information to a user based on the collected data. Such an instrument may employ a transducer to sense changes in a physical parameter such as temperature or pressure, and to convert the sensed information into electrical signals such as voltage or frequency variations. The term "instrument" may also cover a physical or software device that performs an analysis on data acquired from another instrument and then outputs the processed data to display or recording means. This second category of instruments includes oscilloscopes, spectrum analyzers and digital millimeters. The types of source data collected and analyzed by instruments may thus vary widely, including both physical parameters such as temperature, pressure, distance, light and sound frequencies and amplitudes, and also electrical parameters including voltage, current and frequency.

A virtual instrument (VI) is defined as an industry-standard computer equipped with user-friendly application software, cost-effective hardware and driver software that together perform the functions of traditional instruments. Simulated physical instruments are called virtual instruments (VIs). Virtual instrumentation software based on user requirements defines general-purpose measurement and control hardware functionality. With virtual instrumentation, engineers and scientists reduce development time, design higher quality products, and lower their design costs. In test, measurement and control, engineers have used virtual instrumentation to downsize automated test equipment (ATE) while experiencing up to a several times increase in productivity gains at a fraction of the cost of traditional instrument solutions.

Virtual instrumentation is necessary because it is flexible. It delivers instrumentation with the rapid adaptability required for today's concept, product and process design, development and delivery. Only with virtual instrumentation, engineers and scientists can create the user-defined instruments required to keep up with the world's demands. To meet the ever-increasing demand to innovate and deliver ideas and products faster, scientists and engineers are turning to advanced electronics, processors and software. Consider modern cell phones. Most of them contain the latest features of the last generation, including audio, a phone book and text messaging capabilities. New versions include a camera, MP3 player, and Bluetooth networking and Internet browsing.

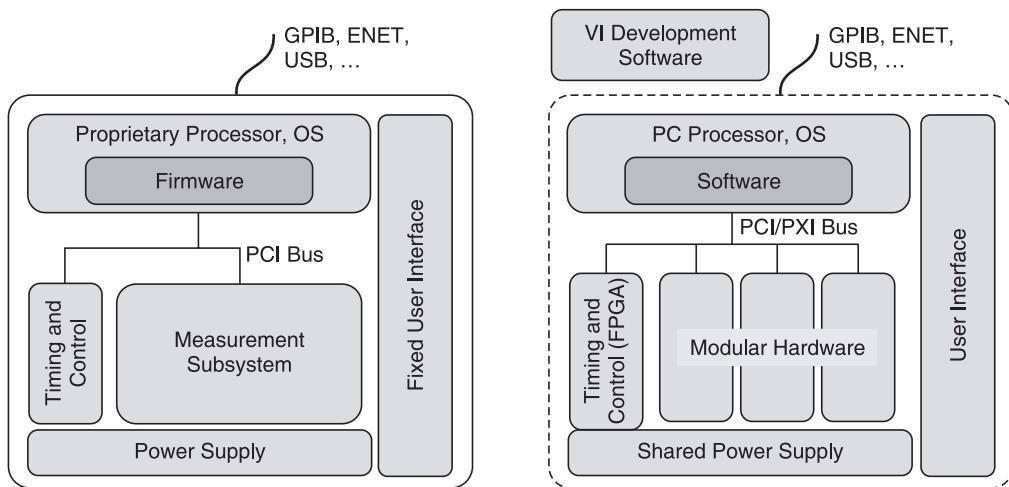
Virtual instruments are defined by the user while traditional instruments have fixed vendor-defined functionality. In a conventional instrument, the set of components that comprise the instrument is fixed and permanently associated with each other. Nevertheless, there is some software that understands these associations. Thus the primary difference between a virtual instrument and a conventional instrument is merely that the associations within a virtual instrument are not fixed but rather managed by software.

Every virtual instrument consists of two parts—software and hardware. A virtual instrument typically has a sticker price comparable to and many times less than a similar traditional instrument for the current measurement task. However, the savings compound over time, because virtual instruments are much more flexible when changing measurement tasks. By not using vendor-defined, prepackaged software and hardware, engineers and scientists get maximum user-defined flexibility. A traditional instrument provides them with all software and measurement circuitry packaged into a product with a finite list of fixed-functionality using the instrument front panel. A virtual instrument provides all the software and hardware needed to accomplish the measurement or control task. In addition, with a virtual instrument, engineers and scientists can customize the acquisition, analysis, storage, sharing and presentation functionality using productive, powerful software.

Without the displays, knobs and switches of a conventional, external box-based instrumentation products, a virtual instrument uses a personal computer for all user interaction and control. In many common measurement applications, a data acquisition board or card, with a personal computer and software, can be used to create an instrument. In fact, a multiple-purpose virtual instrument can be made by using a single data acquisition board or card. The primary benefits of applying data acquisition technology to configure virtual instrumentation include costs, size, and flexibility and ease of programming. The cost to configure a virtual instrumentation-based system using a data acquisition board or cards can be as little as 25% of the cost of a conventional instrument.

Traditional instruments and software-based virtual instruments largely share the same architectural components, but radically different philosophies as shown in Figure 1.11. Conventional

instruments as compared to a virtual instrumentation can be very large and cumbersome. They also require a lot of power, and often have excessive amounts of features that are rarely, if ever used. Most conventional instruments do not have any computational power as compared to a virtual instrument. Since the virtual instrument is part of a person computer configuration, the personal computer's computational as well as controlling capability can be applied into a test configuration. Virtual instruments are compatible with traditional instruments almost without exception. Virtual instrumentation software typically provides libraries for interfacing with common ordinary instrument buses such as GPIB, serial or Ethernet.



**Figure 1.11** Traditional instruments (left) and software based virtual instruments (right).

Except for the specialized components and circuitry found in traditional instruments, the general architecture of stand-alone instruments is very similar to that of a PC-based virtual instrument. Both require one or more microprocessors, communication ports (for example, serial and GPIB), and display capabilities, as well as data acquisition modules. What makes one different from the other is their flexibility and the fact that we can modify and adapt the instrument to our particular needs. A traditional instrument might contain an integrated circuit to perform a particular set of data processing functions; in a virtual instrument, these functions would be performed by software running on the PC processor. We can extend the set of functions easily, limited only by the power of the software used. By employing virtual instrumentation solutions, we can lower capital costs, system development costs, and system maintenance costs, while improving time to market and the quality of our own products.

There is a wide variety of hardware devices available which we can either plug into the computer or access through a network. These devices offer a wide range of data acquisition capabilities at a significantly lower cost than that of dedicated devices. As integrated circuit technology advances, and off-the-shelf components become cheaper and more powerful, so do the boards that use them. With these advances in technology, comes an increase in data acquisition rates, measurement accuracy, precision and better signal isolation. Depending on the particular application, the hardware we choose might include analog input or output, digital input or output, counters, timers, filters, simultaneous sampling, and waveform generation capabilities.

Virtual instrumentation has achieved mainstream adoption by providing a new model for building measurement and automation systems. Keys to its success include rapid PC advancement; explosive low-cost, high-performance data converter (semiconductor) development; and system design software emergence. These factors make virtual instrumentation systems accessible to a very broad base of users.

Virtual instruments take advantage of PC performance increase by analyzing measurements and solving new application challenges with each new-generation PC processor, hard drive, display and I/O bus. These rapid advancements combined with the general trend that technical and computer literacy starts early in school, contribute to successful computer-based virtual instrumentation adoption. The virtual instrumentation driver is the proliferation of high-performance, low-cost analog-to-digital (ADC) and digital-to-analog (DAC) converters. Applications such as wireless communication and high-definition video impact these technologies relentlessly. Virtual instrumentation hardware uses widely available semiconductors to deliver high-performance measurement front ends. Finally, system design software that provides an intuitive interface for designing custom instrumentation systems furthers virtual instrumentation. Various interface standards are used to connect external devices to the computer. PC is the dominant computer system in the world today. VI is supported on the PC under Windows, Linux, Macintosh, Sun, and HP operating systems. All VI platforms provide powerful Graphical User Interfaces (GUIs) for development and implementation of the solutions.

## **1.6 HARDWARE AND SOFTWARE IN VIRTUAL INSTRUMENTATION**

### **1.6.1 Role of Hardware in Virtual Instrumentation**

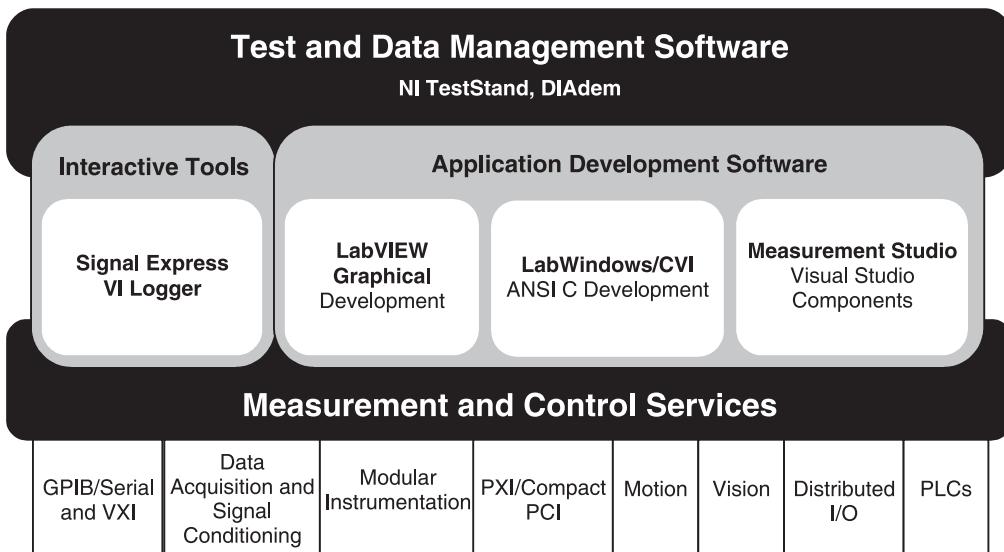
Input/Output plays a critical role in virtual instrumentation. To accelerate test, control and design, I/O hardware must be rapidly adaptable to new concepts and products. Virtual instrumentation delivers this capability in the form of modularity within scalable hardware platforms. Virtual instrumentation is software-based; if we can digitize it, we can measure it. Standard hardware platforms that house the I/O are important to I/O modularity. Laptops and desktop computers provide an excellent platform where virtual instrumentation can make the most of existing standards such as the USB, PCI, Ethernet, and PCMCIA buses.

### **1.6.2 Role of Software in Virtual Instrumentation**

Software is the most important component of a virtual instrument. With the right software tool, engineers and scientists can efficiently create their own applications by designing and integrating the routines that a particular process requires. You can also create an appropriate user interface that best suits the purpose of the application and those who will interact with it. You can define how and when the application acquires data from the device, how it processes, manipulates and stores the data, and how the results are presented to the user. With powerful software, we can build intelligence and decision-making capabilities into the instrument so that it adapts when measured signals change inadvertently or when more or less processing power is required. An important advantage that software provides is modularity. When dealing with a large project, engineers and scientists generally approach the task by breaking it down into functional solvable units. These subtasks are more manageable and easier to test, given the reduced dependencies that might cause

unexpected behaviour. We can design a virtual instrument to solve each of these subtasks, and then join them into a complete system to solve the larger task. The ease with which we can accomplish this division of tasks depends greatly on the underlying architecture of the software.

A virtual instrument is not limited or confined to a stand-alone PC. In fact, with recent developments in networking technologies and the Internet, it is more common for instruments to use the power of connectivity for the purpose of task sharing. Typical examples include supercomputers, distributed monitoring and control devices, as well as data or result visualization from multiple locations. Every virtual instrument is built upon flexible, powerful software by an innovative engineer or scientist applying domain expertise to customize the measurement and control application. The result is a user-defined instrument specific to the application needs. Virtual instrumentation software can be divided into several different layers like the application software, test and data management software, measurement and control services software as shown in Figure 1.12.



**Figure 1.12** Layers of virtual instrumentation software.

Most people think immediately of the application software layer. This is the primary development environment for building an application. It includes software such as LabVIEW, LabWindows/CVI (ANSI C), Measurement Studio (Visual Studio programming languages), Signal Express and VI Logger. Above the application software layer is the test executive and data management software layer. This layer of software incorporates all of the functionality developed by the application layer and provides system-wide data management. Measurement and control services software is equivalent to the I/O driver software layer. It is one of the most crucial elements of rapid application development. This software connects the virtual instrumentation software and the hardware for measurement and control. It includes intuitive application programming interfaces, instrument drivers, configuration tools, I/O assistants and other software included with the purchase of hardware. This software offers optimized integration with both hardware and application development environments.

## 1.7 VIRTUAL INSTRUMENTATION FOR TEST, CONTROL AND DESIGN

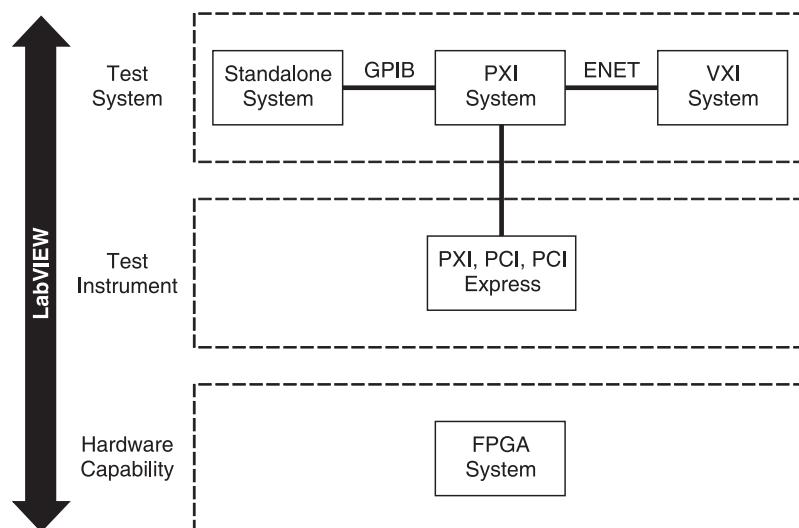
Virtual instrumentation has been widely adopted in test and measurement areas. It has gradually increased addressable applications through continuous innovation and hundreds of measurement hardware devices. The benefits that have accelerated test development are beginning to accelerate control and design.

### 1.7.1 Virtual Instrumentation for Test

Test has been a long-proven field for virtual instrumentation. As the pace of innovation has increased, so too has the pressure to get new, differentiated products to market quickly. Consumer expectations continue to increase; in electronics markets, for example, disparate function integration is required in a small space and at a low cost. All of these conditions drive new validation, verification and manufacturing test needs. A test platform that can keep pace with this innovation is not optional; it is essential. The platform must include rapid test development tools adaptable enough to be used throughout the product development flow. The need to get products to market quickly and manufacture them efficiently requires high-throughput test. To test the complex multifunctional products that consumers demand requires precise, synchronized measurement capabilities. And as companies incorporate innovations to differentiate their products, test systems must quickly adapt to test the new features.

Virtual instrumentation is an innovative solution to these challenges. It combines rapid development software and modular, flexible hardware to create user-defined test systems. Virtual instrumentation delivers as shown in Figure 1.13 user-defined instruments and customizable hardware for test systems:

- Intuitive software tools for rapid test development
- Fast, precise modular I/O based on innovative commercial technologies
- A PC-based platform with integrated synchronization for high accuracy and throughput



**Figure 1.13** User-defined instruments and customizable hardware for test systems.

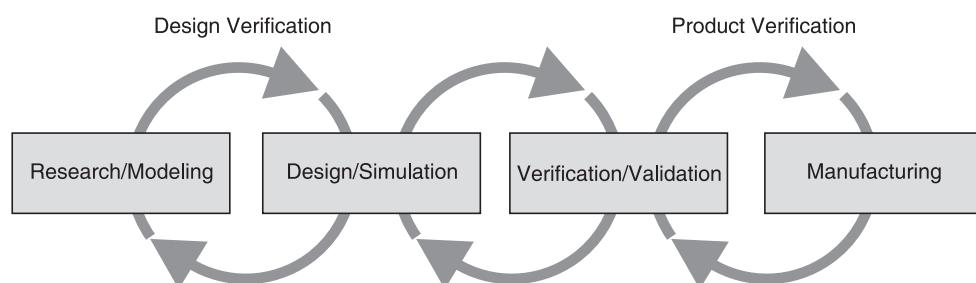
Engineers and scientists have always been able to use virtual instrumentation software to create highly integrated user-defined systems using modular I/O, but they can now extend custom configurability to the hardware itself. This degree of user-configurability and transparency will change the way engineers build test systems.

### 1.7.2 Virtual Instrumentation for Industrial I/O and Control

PCs and PLCs both play an important role in control and industrial applications. PCs bring greater software flexibility and capability, while PLCs deliver outstanding ruggedness and reliability. But as control needs become more complex, there is a recognized need to accelerate the capabilities while retaining the ruggedness and reliabilities. Independent industry experts have recognized the need for tools that can meet the increasing need for more complex, dynamic, adaptive and algorithm-based control. The Programmable Automation Controllers (PACs) provide multidomain functionality (logic, motion, drives and process) and the concept of PAC supports multiple I/O types. Logic, motion and other function integrations are a requirement for increasingly complex control approaches. PACs deliver PC software flexibility with PLC ruggedness and reliability. LabVIEW software and rugged, real-time, control hardware platforms are ideal for creating a PAC.

### 1.7.3 Virtual Instrumentation for Design

The same design engineers that use a wide variety of software design tools must use hardware to test prototypes. Commonly, there is no good interface between the design phase and testing/validation phase, which means that the design usually must go through a completion phase and enter a testing/validation phase. Issues discovered in the testing phase require a design-phase reiteration as shown in Figure 1.14. The simulation test plays a critical role in the design and manufacture of today's electronic devices.



**Figure 1.14** Simulation test plays a critical role in the design and manufacture.

In reality, the development process has two very distinct and separate stages—design and test—which are two individual entities. On the design side, Electronic Design Automation (EDA) tool vendors undergo tremendous pressure to interoperate from the increasing semiconductor design and manufacturing group complexity requirements. Engineers and scientists are demanding the capability to reuse designs from one tool in other tools as products go from schematic design to simulation to physical layout. Similarly, test system development is evolving towards a modular approach. The gap between these two worlds has traditionally been neglected, first noticeable in

the new product prototype stage. Traditionally, this is the stage where the product designer uses benchtop instruments to sanity-check the physical prototypes against their design for correctness. The designer makes these measurements manually, probing circuits and looking at the signals on instruments for problems or performance limitations. As designs iterate through this build-measure-tweak-rebuild process, the designer needs the same measurements again. In addition, these measurements can be complex—requiring frequency, amplitude and temperature sweeps with data collected and analyzed throughout. Because these engineers focus on design tools, they are reluctant to invest in learning to automate their testing. Systems with intrinsic-integration properties are easily extensible and adapt to increasing product functionality. When new tests are required, engineers simply add new modules to the platform to make the measurements. Virtual instrumentation software flexibility and virtual instrumentation hardware modularity make virtual instruments a necessity to accelerate the development cycle.

## 1.8 VIRTUAL INSTRUMENTATION IN THE ENGINEERING PROCESS

Virtual instruments provide significant advantages in every stage of the engineering process—from research and design to manufacturing test.

### 1.8.1 Research and Development

In research and design, engineers and scientists demand rapid development and prototyping capabilities. With virtual instruments, we can quickly develop a program, take measurements from an instrument to test a prototype and analyze results—all in a fraction of the time required to build tests with traditional instruments. When we need flexibility, a scalable open platform is essential, from the desktop to embedded systems to distributed networks. The demanding requirements of research and development (R&D) applications require seamless software and hardware integration. Whether we need to interface stand-alone instruments using GPIB or directly acquire signals into the computer with a data acquisition board and signal conditioning hardware, virtual instrumentation software makes integration simple. With virtual instruments, we also can automate a testing procedure, eliminating the possibility of human error and ensuring the consistency of the results by not introducing unknown or unexpected variables.

### 1.8.2 Development Test and Validation

With the flexibility and power of virtual instruments, one can easily build complex test procedures. For automated design verification testing, one can create test routines in virtual instrumentation software and integrate software such as National Instruments Test Stand, which offers powerful test management capabilities. One of the many advantages these tools offer across the organization is code reuse. We develop code in the design process and then plug these same programs into functional tools for validation, test or manufacturing.

### 1.8.3 Manufacturing Test

Decreasing test time and simplifying development of test procedures are primary goals in manufacturing test. Virtual instruments combined with powerful test management software deliver

high performance. These tools meet rigorous throughput requirements with a high-speed, multithreaded engine for running multiple test sequences in parallel. TestStand easily manages test sequencing, execution and reporting based on routines written in virtual instrumentation software. TestStand integrates the creation of test code in virtual instrumentation software. TestStand also can reuse code created in research and development or design and validation. If we have manufacturing test applications, we can take full advantage of the work already done in the product life cycle.

Manufacturing applications require software to be reliable, high in performance and interoperable. Virtual instruments offer all these advantages by integrating features such as alarm management, historical data trending, security, networking, industrial I/O and enterprise connectivity. With this functionality, we can easily connect to many types of industrial devices such as PLCs, industrial networks, distributed I/O and plug-in data acquisition boards. By sharing code across the enterprise, manufacturing can use the same applications developed in research and development or validation, and integrate seamlessly with manufacturing test processes.

## **1.9 VIRTUAL INSTRUMENTS BEYOND PERSONAL COMPUTER**

Recently, commercial PC technologies have begun migrating into embedded systems. Examples include Windows CE, Intel x86-based processors, PCI and Compact PCI buses and Ethernet for embedded development. Virtual instrumentation relies on commercial technologies for cost and performance advantages; it has also expanded to encompass more embedded and real-time capabilities. LabVIEW runs on Linux as well as the embedded ETS real-time operating system from VenturCom on specific embedded targets. The option of using virtual instrumentation as a scalable framework that extends from the desktop to embedded devices should be considered a tool in the complete toolbox of an embedded systems developer. Ethernet now dominates as the standard network infrastructure for companies worldwide. In addition, the popularity of the Web interface in the PC world has overflowed into the development of cell phones, PDAs, and now industrial data acquisition and control systems. Embedded systems at one time meant stand-alone operation, or at most interfacing at a low level with a real-time bus to peripheral components. The increased demand for information at all levels of the enterprise (and in consumer products) requires you to network embedded systems while continuing to guarantee reliable and often real-time operation.

Virtual instrumentation software can combine one development environment for both desktop and real-time systems using cross-platform compiled technology; you can capitalize on the built-in Web servers and easy-to-use networking functionality of desktop software and target it to real-time and embedded systems. For example, you can use LabVIEW to simply configure a built-in Web server to export an application interface to defined secure machines on the network on Windows, and then download that application to run on a headless embedded system that can fit in the user's hand. This procedure happens with no additional programming required on the embedded system. You then can deploy that embedded system, power it, connect to the application from a remote secure machine via Ethernet, and interface to it using a standard Web browser. For more sophisticated networking applications, you can graphically program TCP/IP or other methods with which you are already familiar in LabVIEW and then run them in the embedded system.

Embedded systems development is one of the fastest growing segments of engineering, and will continue to be for the foreseeable future as consumers demand smarter cars, appliances, homes, and so on. The evolution of these commercial technologies will propel virtual instrumentation into being more applicable to a growing number of applications. Leading companies that provide virtual instrumentation software and hardware tools need to invest in expertise and product development to serve this growing set of applications. Virtual instrumentation software platform, LabVIEW, includes the ability to scale from development for desktop operating systems, to embedded real-time systems to handheld personal digital assistant targets, to FPGA-based hardware, and even to enabling smart sensors.

Next-generation virtual instrumentation tools need to include networking technology for quick and easy integration of Bluetooth, wireless Ethernet and other standards. In addition to using these technologies, virtual instrumentation software needs a better way to describe and design timing and synchronization relationships between distributed systems in an intuitive way to help faster development and control of these often embedded systems. The virtual instrumentation concepts of integrated software and hardware, flexible modular tools, and the use of commercial technologies combine to create a framework upon which we can rapidly complete our systems development and also maintain them for the long term. Because virtual instrumentation offers so many options and capabilities in embedded development, it makes sense for embedded developers to understand and review these tools.

## **1.10 GRAPHICAL SYSTEM DESIGN USING LabVIEW**

The development of VI is linked to the evolution of small inexpensive personal computers and progress in computer hardware and software. Graphical User Interfaces (GUIs) too played a vital role in the development of VI. While the original application of replacing the controls with a computer link used a command line interface, the use of sophisticated GUI became an essential component in the later versions of VI software. The major developments that helped VI become successful were development of low cost computer systems of adequate computing power, evolution of good GUI, development of standards for buses, and increasingly stable networking platforms. The computing power of the processors improved by leaps and bounds permitting the development of all these resource intensive techniques. Standardization and progress in computing hardware increased. Developments of various interfacing standards lead to a massive reduction in development effort. All these developments have occurred in parallel and have taken us to the situation where VI is the dominant tool for the development and implementation of instrumentation applications and systems.

The term “VI” owes a lot to the development of the **Laboratory Virtual Instrument Engineering Workbench (LabVIEW)** by National Instruments, Austin, Texas, USA. In 1983, National Instruments began to search for a way to minimize the time needed to program instrumentation systems. LabVIEW made its appearance as an interpreted package on the Apple Macintosh in 1986. LabVIEW 2 was released in 1990, and was a compiled package as against an interpreted package. The first reasonably stable graphical environment (Windows 3.0) made its appearance only in 1990.

In 1992, they introduced LabVIEW for Windows and LabVIEW for Sun based on the new portable architecture. LabVIEW 3 arrived in 1993 for Macintosh, Windows and Sun operating

systems. LabVIEW 3 programs written on one platform could run on another. In 1999, LabVIEW became available for the Linux platform as well. LabVIEW 4, released in 1996, featured a more customizable development environment so that users could create their own workspace to match their industry, experience level and development habits. In addition, LabVIEW 4 added high-powered editing and debugging tools for advanced instrumentation systems, as well as OLE-based connectivity and distributed execution tools.

Networking support on smaller systems was first introduced in Version 5. LabVIEW 5 and 5.1 (in 1999) continued to improve on the development tool by introducing a built-in Web server, a dynamic programming and control framework (VI server), integration with ActiveX, and easy sharing of data over the Internet with a protocol called DataSocket. In 2000, LabVIEW 6 (sometimes called 6i) provided both an easy and intuitive programming interface. In 2001, LabVIEW 6.1 introduced event-oriented programming, remote Web control of LabVIEW.

Version 7 has expanded the horizon to make it simpler for the inexperienced user by providing various Express utilities and Assistants. LabVIEW has two closely related products—BridgeVIEW and LabVIEW RT (for realtime applications). BridgeVIEW can also be called LabVIEW industrial. The RT module of LabVIEW was originally designed to support distributed computing and real-time applications. LabVIEW RT is a hardware and software combination that allows you to take portions of your LabVIEW code and download them to be executed on a separate controller board with its own real-time operating system. In certain respects LabVIEW RT and the development of FPGA (Field Programmable Gate Array) support in LabVIEW overlap. It is possible to program FPGA modules to carry out many tasks which may be assigned to RT systems.

National Instruments released LabVIEW 8.5, the latest version of the graphical system design platform for test, control and embedded system development. LabVIEW 8.5 simplifies multicore as well as FPGA-based application development with its intuitive parallel dataflow language. NI LabVIEW 8.6 includes the platform installation DVDs, Block Diagram Cleanup, Quick Drop and Web services creation. In addition to these added tools to the LabVIEW development system, we can explore key new features in the LabVIEW modules and toolkits, including industrial function blocks in the LabVIEW Real-Time Module and new LabVIEW FPGA Module IP.

With the enhanced releases of LabVIEW graphical programming software, system-level engineers as well as domain experts with little to no embedded expertise can accurately work with systems of increased complexity and scale, thereby, drastically reducing the time from concept to prototype.

## **1.11 GRAPHICAL PROGRAMMING AND TEXTUAL PROGRAMMING**

Graphical programming is a visually-oriented approach to programming. Graphical programming is easier and more intuitive to use than traditional textual programming. Textual programming requires the programmers to be reasonably proficient in the programming language. Non-programmers can easily learn the graphical approach faster at less amount of time.

The main advantage of textual languages like C is that they tend to have faster graphical approach execution time and better performance than graphical programs. Textual programming environments are typically used in determining high throughput virtual instrumentation systems, such as manufacturing test systems.

Textual programming environments are popular and many engineers are trained to use these standardized tools. Graphical environments are better for nonprogrammers and useful for developing virtual instruments quickly and need to be reconfigured rapidly.

Virtual instrumentation is not limited to graphical programming but can be implemented using a conventional programming language. In R & D design characterization or in a troubleshooting situation Microsoft Excel (spreadsheet) or Microsoft Word (word processor) can be used. The most important task is to understand how to use standard analysis packages that can directly input data from the instruments and can be used to analyze, store and present the information in a useful format. Irrespective of whether it is classical or graphical environment any system with a graphical system design can be looked at as being composed of two parts—the user interface and the underlying code. The code in a conventional language like C comprises a number of routines while in the graphical language G it is a collection of icons interconnected by multi-colored lines. Table 1.1 compares text-based programming and graphical programming.

**TABLE 1.1** Comparison of text-based and graphical programming

Text-based programming	Graphical programming
<ul style="list-style-type: none"><li>• Syntax must be known to do programming.</li><li>• The execution of the program is from <i>top to bottom</i>.</li><li>• To check for the <i>error</i> the program has to be compiled or executed.</li><li>• <i>Front panel</i> design needs extra coding or needs extra work.</li><li>• Text-based programming is <i>non interactive</i>.</li><li>• This is text-based programming where the programming is a <i>conventional method</i>.</li><li>• <i>Logical Error</i> finding is easy in large programs.</li><li>• Program flow is <i>not visible</i>.</li><li>• It is <i>test-based</i> programming.</li><li>• Passing parameters to <i>sub routine</i> is difficult.</li></ul>	<ul style="list-style-type: none"><li>• Syntax is knowledge but is not required for programming.</li><li>• The execution of program is from <i>left to right</i>.</li><li>• Errors are indicated as we wire the blocks.</li><li>• <i>Front panel</i> design is a part of programming.</li><li>• Graphical programming is highly <i>interactive</i>.</li><li>• The programming is <i>Data Flow Programming</i>.</li><li>• <i>Logical Error</i> finding in large programs is quiet complicated.</li><li>• Data flow is <i>visible</i>.</li><li>• It is <i>icon-based</i> programming and wiring.</li><li>• Passing parameters to <i>sub VI</i> is easy.</li></ul>

---

## SUMMARY

---

- Graphical system design is a revolutionary approach to solving design challenges of an entire system that blends graphical programming and flexible commercial-off-the-shelf (COTS) hardware.
- The functionality of graphical system design is design, prototype and deploy.
- Simulated physical instruments are called Virtual Instruments or VIs. VIs are programs created using LabVIEW software.
- The advantage of virtual instruments is that instruments can be defined inside the software.
- Characteristics of virtual instruments are lower costs of instrumentation, portability between various computer platforms, easy-to-use graphical user interface, graphical representation

of program structures, and code can be compiled to standalone. EXE or .DLL file, TCP/IP connectivity (Web server integrated into virtual instrument)

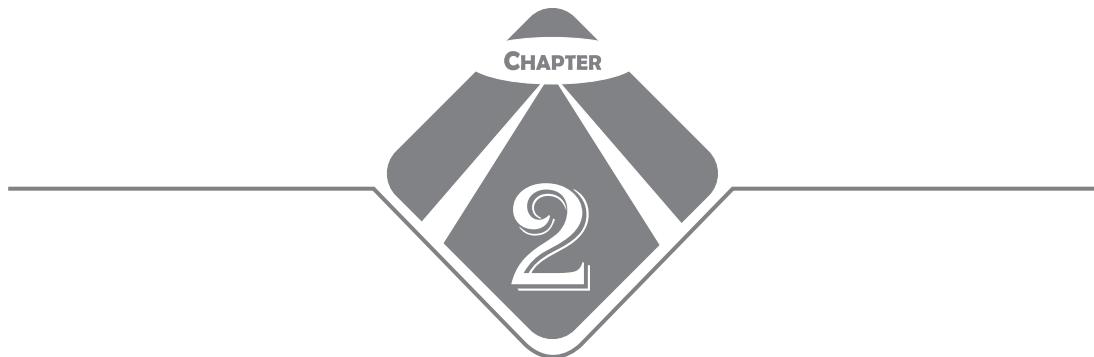
- Many engineers and scientists successfully use graphical programming as their primary tool in a wide range of applications, from design to deployment of machines, remote monitoring, embedded systems and other complex systems.
- Graphical programming environments are easier to use and require less expertise than textual programming environments.

---

### REVIEW QUESTIONS

---

1. What is graphical system design? With a neat block diagram explain its functionalities.
2. Draw and explain the virtual instrumentation model and graphical system design model.
3. Draw a block diagram of a typical embedded system software and hardware design flow and compare with stream-lined development flow with graphical system design.
4. Explain with a block diagram the general basic components of high level instruments.
5. What is virtual instrumentation?
6. Draw and explain the basic difference between the traditional instruments and software-based virtual instruments.
7. Draw and explain the layers of virtual instrumentation software and the software role.
8. Explain with a block diagram how simulation test plays a critical role in the design and manufacture of a product.
9. Compare text-based programming and graphical programming.



# INTRODUCTION TO LabVIEW

---

## 2.1 INTRODUCTION

LabVIEW (**L**aboratory **V**irtual **I**nstrument **E**ngineering **W**orkbench) is a graphical programming environment which has become prevalent throughout research labs, academia and industry. It is a powerful and versatile analysis and instrumentation software system for measurement and automation. Its graphical programming language called *G programming* is performed using a graphical block diagram that compiles into machine code and eliminates a lot of the syntactical details. LabVIEW offers more flexibility than standard laboratory instruments because it is software based. Using LabVIEW, the user can originate exactly the type of virtual instrument needed and programmers can easily view and modify data or control inputs. The popularity of the National Instruments LabVIEW graphical dataflow software for beginners and experienced programmers in so many different engineering applications and industries can be attributed to the software's intuitive graphical programming language used for automating measurement and control systems.

LabVIEW programs are called virtual instruments (VIs), because their appearance and operation imitate physical instruments like oscilloscopes. LabVIEW is designed to facilitate data collection and analysis, as well as offers numerous display options. With data collection, analysis and display combined in a flexible programming environment, the desktop computer functions as a dedicated measurement device. LabVIEW contains a comprehensive set of VIs and functions for acquiring, analyzing, displaying, and storing data, as well as tools to help you troubleshoot your code.

All test, measurement and control applications can be divided into three main components and the key to virtual instrumentation is the ability to acquire, analyze and present data. LabVIEW can acquire data using the devices like GPIB, Serial, Ethernet, VXI, PXI Instruments, Data Acquisition (DAQ), PCI *eXtensions* for Instrumentation (PXI), Image Acquisition (IMAQ), Motion Control, Real-Time (RT) PXI, PLC (through OPC Server), PDA, and Modular Instruments. To help you analyze your data LabVIEW includes analysis functions for Differential Equations, Optimization, Curve Fitting, Calculus, Linear Algebra, Statistics and so on. Express VIs are specifically designed for measurement analysis, including filtering and spectral analysis. Signal Processing VIs for Filtering, Windowing, Transforms, Peak Detection, Harmonic Analysis, and Spectrum Analysis are provided. LabVIEW includes the following tools to help in presenting data on the computer; Graphs, Charts, Tables, Gauges, Meters, Tanks, 3D Controls, Picture Control, 3D Graphs and Report Generation. Over the Internet, Web Publishing Tools, Data socket (Windows Only), TCP/IP, VI Server, Remote Panels and Email are available to present data.

LabVIEW can communicate with hardware such as data acquisition, vision, and motion control devices, and GPIB, PXI, VXI, RS-232, and RS-485 devices. LabVIEW also has built-in features for connecting your application to the Web using the LabVIEW Web Server and software standards such as TCP/IP networking and ActiveX. Using LabVIEW, you can create test and measurement, data acquisitions, instrument control, datalogging, measurement analysis, and report generation applications. You also can create stand-alone executables and shared libraries, like DLLs, because LabVIEW is a true 32-bit compiler.

For new programmers, LabVIEW Express technology transforms common measurement and automation tasks into much higher-level, intuitive VIs. With Express technology, thousands of nonprogrammers have taken advantage of the LabVIEW platform to build automated systems quickly and easily. For experienced programmers, LabVIEW delivers the performance, flexibility and compatibility of a traditional programming language such as C or BASIC. In fact, the full-featured LabVIEW programming language has the same constructs that traditional languages have such as variables, data types, objects, looping and sequencing structures, as well as error handling. And with LabVIEW, programmers can reuse legacy code packaged as DLLs or shared libraries and integrate with other software using ActiveX, TCP and other standard technologies. The LabVIEW Family consists of NI LabVIEW Graphical Programming Software for Measurement and Automation, LabVIEW Real-Time Module, LabVIEW FPGA Module, LabVIEW PDA Module, LabVIEW Datalogging and Supervisory Control Module.

## 2.2 ADVANTAGES OF LabVIEW

The following are the advantages of LabVIEW:

- **Graphical user interface:** Design professionals use the drag-and-drop user interface library by interactively customizing the hundreds of built-in user objects on the controls palette.
- **Drag-and-drop built-in functions:** Thousands of built-in functions and IP including analysis and I/O, from the functions palette to create applications easily.

- **Modular design and hierarchical design:** Run modular LabVIEW VIs by themselves or as subVIs and easily scale and modularize programs depending on the application.
- **Multiple high level development tools:** Develop faster with application specific development tools, including the LabVIEW Statechart Module, LabVIEW Control Design and Simulation Module and LabVIEW FPGA Module.
- **Professional Development Tools:** Manage large, professional applications and tightly integrated project management tools; integrated graphical debugging tools; and standardized source code control integration.
- **Multi platforms:** The majority of computer systems use the Microsoft Windows operating system. LabVIEW works on other platforms like Mac OS, Sun Solaris and Linux. LabVIEW applications are portable across platforms.
- **Reduces cost and preserves investment:** A single computer equipped with LabVIEW is used for countless applications and purposes—it is a versatile product. Complete instrumentation libraries can be created for less than the cost of a single traditional, commercial instrument.
- **Flexibility and scalability:** Engineers and scientists have needs and requirements that can change rapidly. They also need to have maintainable, extensible solutions that can be used for a long time. By creating virtual instruments based on powerful development software such as LabVIEW, you inherently design an open framework that seamlessly integrates software and hardware. This ensures that your applications not only work well today but that you can easily integrate new technologies in the future.
- **Connectivity and instrument control:** LabVIEW has ready-to-use libraries for integrating stand-alone instruments, data acquisition devices, motion control and vision products, GPIB/IEEE 488 and serial/RS-232 devices, and PLCs to build a complete measurement and automation solution. Plug-and Play instrument drivers access the industry's largest source of instrument drivers with several instruments from various vendors.
- **Open environment:** LabVIEW provides the tools required for most applications and is also an open development environment. This open language takes advantage of existing code; can easily intergrate with legacy systems and incorporate third party software with .NET, ActiveX, DLLs, objects, TCP, Web services and XML data formats.
- **Distributed development:** Can easily develop distributed applications with LabVIEW, even across different platforms. With powerful server technology you can offload processor-intensive routines to other machines for faster execution, or create remote monitoring and control applications.
- **Visualization capabilities:** LabVIEW includes a wide array of built-in visualization tools to present data on the user interface of the virtual instrument as chart, graphs, 2D and 3D visualization. Reconfiguring attributes of the data presentation, such as colours, font size, graph types, and more can be easily performed.
- **Rapid development with express technology:** Use configuration-based Express VIs and I/O assistants to rapidly create common measurement applications without programming by using LabVIEW Signal Express.

- **Compiled language for fast execution:** LabVIEW is a compiled language that generates optimized code with execution speeds comparable to compiled C and develops high-performance code.
- **Simple application distribution:** Use the LabVIEW application builder to create executables(exes) and shared libraries (DLLs) for deployment.
- **Target management:** Easily manage multiple targets, from real-time to embedded devices including FPGAs, microprocessors, microcontrollers, PDAs and touch panels.
- **Object-oriented design:** Use object-oriented programming structures to take advantage of encapsulation and inheritance to create modular and extensible code.
- **Algorithm design:** Develop algorithms using math-oriented textual programming and interactively debug .m file script syntax with LabVIEW MathScript.

## 2.3 SOFTWARE ENVIRONMENT

There are three steps to create our application in the software environment:

- Design a user interface
- Draw our graphical code
- Run our program

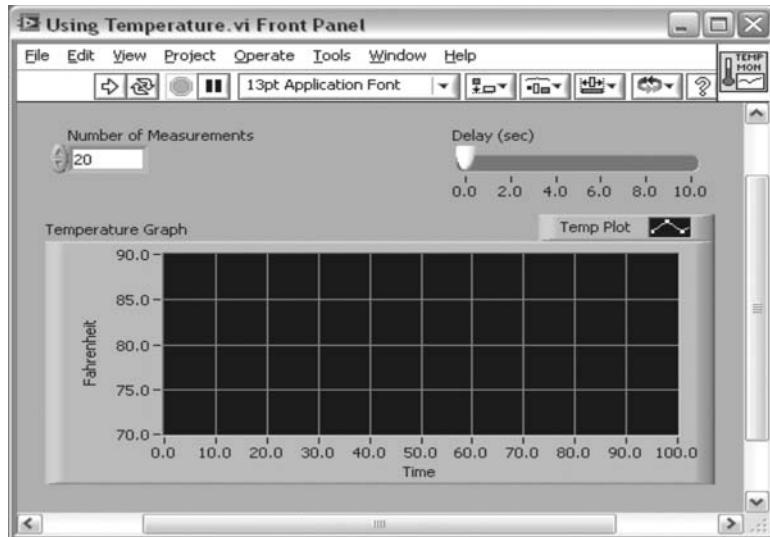
A virtual instrument (VI) has three main components—the front panel, the block diagram and the icon/connector pane. In order to use a VI as a subVI in the block diagram of another VI, it is essential that it contains an icon and a connector. The two LabVIEW windows are the front panel (containing controls and indicators) and block diagram (containing terminals, connections and graphical code). The front panel is the user interface of the virtual instrument. The code is built using graphical representations of functions to control the front panel objects. The block diagram contains this graphical source code. In LabVIEW, you build a user interface or front panel with controls and indicators. Controls are knobs, push buttons, dials and other input devices. Indicators are graphs, LEDs and other displays. After you build the user interface, you can add code using VIs and structures to control the front panel objects. The block diagram contains this code. In some ways, the block diagram resembles a flowchart.

### 2.3.1 Front Panel Windows

When you open a new or existing VI, the front panel of the VI appears. The front panel is the interactive user interface for the VI. It is named a front panel because it stimulates the front panel of a physical instrument. Build the front panel with controls and indicators as shown in Figure 2.1.

One of the most powerful features that LabVIEW offers engineers and scientists is its graphical programming environment to design custom virtual instruments by creating a graphical user interface on the computer screen to

- Operate the instrumentation program
- Control selected hardware
- Analyze acquired data
- Display results

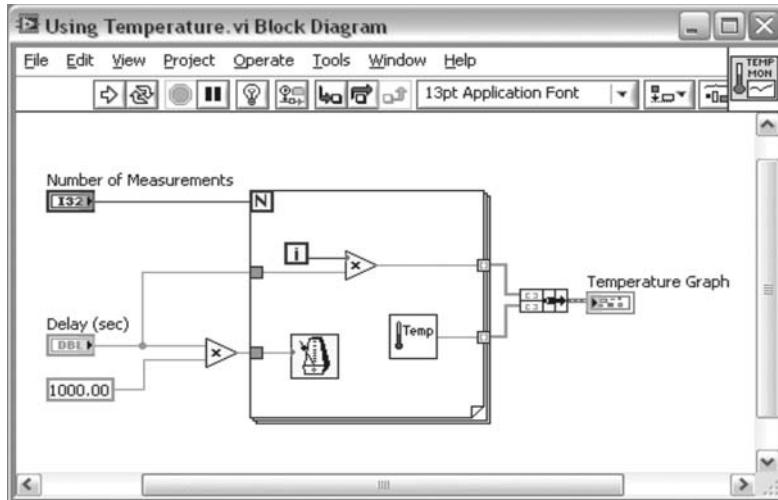


**Figure 2.1** LabVIEW virtual instrument front panel.

The front panel can include knobs, push buttons, graphs and various other controls (which are user inputs) and indicators (which are program outputs). Controls are inputs used to simulate instrument input devices and supply data to the block diagram of the VI, and indicators are outputs displays used to simulate instrument output devices and display data the block diagram acquires or generates. The front panel is customized to emulate control panels of traditional instruments, create custom test panels, or visually represent the control and operation of processes.

### 2.3.2 Block Diagram Windows

The block diagrams accompany the program for the front panel. Front panel objects appear as terminals on the block diagram and the components wired together. After the front panel is built, codes are added using graphical representations of functions in the block diagram to control the front panel objects. The block diagram contains the graphical source code composed of nodes, terminals, and wires. The block diagram is the actual executable program as shown in Figure 2.2. The components of a block diagram are lower-level VIs, built-in functions, constants and program execution control structures. Wires have to be drawn to connect the corresponding objects together to indicate the flow of data between each of them. Front panel objects have analogous terminals on the block diagram so that data can pass easily from the user to the program and back to the user. Use Express VIs, standard VIs and functions on the block diagram to create our measurement code. Block diagram objects include the terminals, subVIs, functions, constants, structures and wires. LabVIEW is the easiest, most powerful tool for acquiring, analyzing and presenting real-world data. Terminals are entry and exit ports that exchange information between the panel and the diagram. Terminals are analogous to parameters and constants in text-based programming languages. Right-click the block diagram objects and select *View As* icon to change the icon view.



**Figure 2.2** LabVIEW virtual instrument block diagram.

### 2.3.3 Icon/Connector Pane

To use a VI as a subVI, it must have an icon and a connector pane. Every VI displays an icon in the upper-right corner of the front panel and block diagram windows. An icon is a graphical representation of a VI. The icon can contain both text and images. To use a VI as a subVI, you need to build a connector pane. The connector pane is a set of terminals that correspond to the controls and indicators of that VI.

## 2.4 CREATING AND SAVING A VI

When you launch LabVIEW, the *Getting Started* window appears as shown in Figure 2.3. You can begin in LabVIEW by starting from a blank VI or project, opening an existing VI or project and modifying it, or opening a template from which to begin your new VI or project.

To open a new project from the *Getting Started* window, select the *Empty Project* option. A new, unnamed project opens, and you can add files to and save the project.

To open a new, blank VI that is not associated with a project, select the *Blank VI* option on the *Getting Started* window. A blank VI opens a blank front panel and blank block diagram.

To Create a VI or Project from a Template select *File»New* to display the *New* dialog box, which lists the built-in VI templates. You also can display the *New* dialog box by clicking the *New* link in the *Getting Started* window.

To open an existing VI select the *Browse* option in the *Getting Started* window to navigate to and open an existing VI.

The menus at the top of a VI window contain items common to other applications, such as *Open*, *Save*, *Copy* and *Paste*, and other items specific to LabVIEW. To save a new VI, select *File»Save*. If you have already saved your VI, select *File»Save As* to access the *Save As* dialog box to create a copy of the VI, or delete the original VI and replace it with the new one.

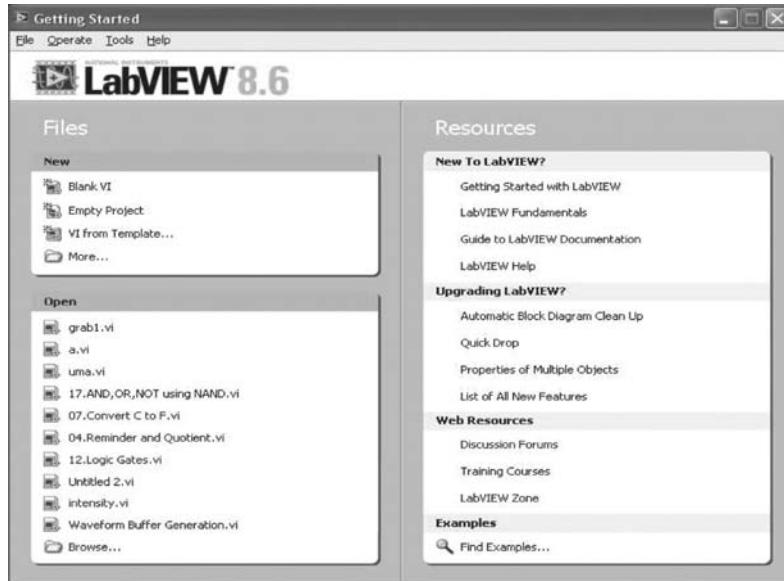


Figure 2.3 Getting Started window.

## 2.5 FRONT PANEL TOOLBAR

Table 2.1 provides the list of front panel toolbars buttons. Click the following toolbar buttons and pull-down menu to run and edit a VI.

TABLE 2.1 Front panel toolbar buttons

	<i>Run</i> button: To run a VI and it appears as a solid white arrow.
	While the VI runs, the <i>Run</i> button appears as shown.
	If the VI that is running is a subVI, the <i>Run</i> button appears as shown.
	<i>Broken Run</i> arrow means a nonexecutable VI and the VI you are creating or editing contains errors. Click this button to display the <i>Error list</i> window, which lists all errors and warnings.
	<i>Run Continuously</i> button: To run the VI until you abort or pause execution. Click the button again to disable continuous running.
	<i>Abort Execution</i> button: To stop the VI immediately if there is no other way to stop the VI.
	<i>Pause</i> button: To pause a running VI.

(Contd.)

**TABLE 2.1** (Contd.)

	<i>Text Settings:</i> To change the font settings including size, style, colour.
	<i>Align Objects:</i> To align objects along axes, including vertical, top edge, left.
	<i>Distribute Objects:</i> To space objects evenly, including gaps, compression.
	<i>Resize Objects:</i> To resize multiple front panel objects to the same size.
	<i>Reorder:</i> When you have objects that overlap each other and you want to define which one is in front or back of another, select one of the objects with the positioning tool and then select from <i>Move Forward</i> , <i>Move Backward</i> , <i>Move To Front</i> , and <i>Move To Back</i> .
	<i>Show Context Help Window</i> button: To toggle the display of the Context Help window.
	<i>Type:</i> Remind you that a new value is available to replace an old value. The <i>Enter</i> button disappears when you click it, press the <Enter> key or click the front panel or block diagram workspace.

## 2.6 BLOCK DIAGRAM TOOLBAR

Table 2.2 provides the details of the block diagram toolbar buttons. Click the following buttons on the block diagram toolbar to debug the VI.

**TABLE 2.2** Block diagram toolbar buttons

	<i>Highlight Execution</i> button: To display an animation of the block diagram execution when you click the <i>Run</i> button. See the flow of data through the block diagram. Click the button again to disable execution highlighting.
	<i>Retain Wire Values:</i> To save the wire values at each point in the flow of execution so that when you place a probe on the wire, you can immediately retain the most recent value of the data that passed through the wire. You must successfully run the VI at least once before you are able to retain the wire values.
	<i>Step Into:</i> To open a node and pause. When you click the <i>Step Into</i> button again, it executes the first action and pauses at the next action of the subVI or structure. You also can press <Ctrl> and down arrow keys. Single-stepping through a VI steps through the VI node by node. Each node blinks to denote when it is ready to execute. By stepping into the node, you are ready to single-step inside the node.
	<i>Step Into:</i> To open a node and pause. When you click the <i>Step Into</i> button again, it executes the first action and pauses at the next action of the subVI or structure. You also can press <Ctrl> and down arrow keys. Single-stepping through a VI steps through the VI node by node. Each node blinks to denote when it is ready to execute. By stepping into the node, you are ready to single-step inside the node.

(Contd.)

**TABLE 2.2** (Contd.)

	<i>Step Over:</i> To execute a node and pause at the next node. You also can press <Ctrl> and right arrow keys. By stepping over the node, you execute the node without single-stepping through the node.
	<i>Step Out:</i> To finish executing the current node and pause. When the VI finishes executing, the <b>Step Out</b> button becomes dimmed. You also can press <Ctrl> and up arrow keys. By stepping out of a node, you can complete single-stepping through the node and navigate to the next node.
	<i>Warning:</i> Appears if a VI includes a warning and you placed a checkmark in the <i>Show Warnings</i> checkbox in the <i>Error List</i> window. A warning indicates there is a potential problem with the block diagram, but it does not stop the VI from running.

## 2.7 PALETTES

LabVIEW has graphical, floating palettes help to create and run VIs. The three palettes are the Tools, Controls, and Functions palettes.

### 2.7.1 Tools Palette

The *Tools* palette shown in Figure 2.4 is available on both the front panel and the block diagram. You can create, modify, and debug VIs using the tools located on the floating *Tools* palette. A tool is a special operating mode of the mouse cursor. The cursor corresponds to the icon of the tool selected in the *Tools* palette. Use the tools to operate and modify the front panel and block diagram objects. You can manually choose the tool you need by selecting it on the *Tools* palette. Or select *View»Tools Palette* to display the *Tools* palette. Also you can press the <Shift> key and right-click to display a temporary version of the *Tools* palette at the location of the cursor. Table 2.3 provides the detailed function of each icon of the tools palette.



**Figure 2.4** Tools palette.

**TABLE 2.3** Tools palette icons

	<i>Automatic Tool Selection button:</i> To enable automatic tool selection. If automatic tool selection is enabled and you move the cursor over objects on the front panel or block diagram, LabVIEW automatically selects the corresponding tool from the <i>Tools</i> palette.
	<i>Operating tool:</i> To change the values of a control or select the text within a control. This tool changes to the icon shown at right when it moves over a text control, such as a numeric or string control.
	<i>Positioning tool:</i> To select, move, or resize objects and it changes to resizing handles when it moves over the edge of a resizable object.

(Contd.)

**TABLE 2.3** (*Contd.*)

	<i>Labeling tool:</i> To edit text and create free labels. This tool changes to the icon on the right when you create free labels.
	<i>Wiring tool:</i> To wire objects together on the block diagram.
	<i>Object Shortcut Menu tool:</i> To access an object shortcut menu with the left mouse button.
	<i>Scrolling tool:</i> To scroll through windows without using scrollbars.
	<i>Breakpoint tool:</i> To set breakpoints on VIs, functions, nodes, wires, and structures to pause execution at that location.
	<i>Probe tool:</i> To create probes on wires on the block diagram. Use the <i>Probe</i> tool to check intermediate values in a VI that produces questionable or unexpected results.
	<i>Color Copy tool:</i> To copy colors for pasting with the <i>Coloring tool</i> .
	<i>Coloring tool:</i> To color an object. It also displays the current foreground and background color settings.

### 2.7.2 Front Panel—Controls Palette

The *Controls* palette shown in Figure 2.5 is available only on the front panel. The *Controls* palette contains the controls and indicators which you can use to create the front panel. The *Controls* palette can be accessed from the front panel by selecting *View»Controls Palette* or by right-clicking an open space on the front panel window to display the Controls palette. The *Controls* palettes contain subpalettes of objects which you can use to create a VI. When you click a subpalette icon, the entire palette changes to the subpalette you selected. To use an object on the palettes, click the object and place it on the front panel.

**Figure 2.5** Controls palette.

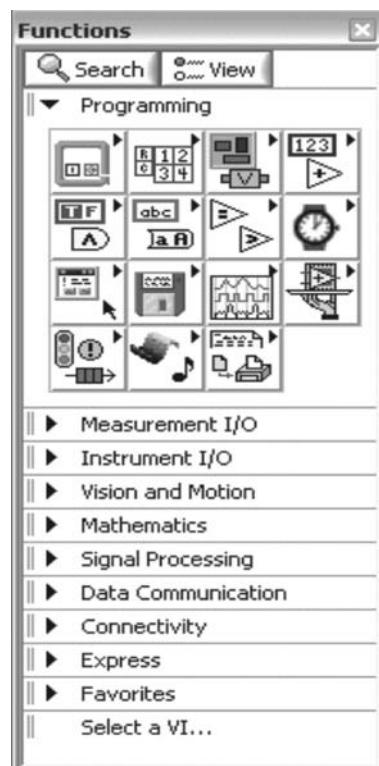
Table 2.4 lists the tools available in the toolbar of the *Controls* and *Function* palettes.

**TABLE 2.4** Controls and Function palette tools

	Use the <i>search</i> button on the <i>Controls and Functions palettes</i> to search for controls, VIs, and functions. In search mode, you can perform text-based searches.
	Use the <i>Options</i> button on the <i>Controls or Functions palette</i> toolbar to change to another palette view or format.
	<i>Up to Owning palette</i> —Navigates up one level in the palette hierarchy.

### 2.7.3 Block Diagram—Functions Palette

Use the *Functions* palette to create the block diagram. The *Functions* palette as shown in Figure 2.6 contains the sub VIs, functions, and constants that are available only on the block diagram. You can access the *Functions* palette from the block diagram by selecting *View»Functions Palette*. You can also right-click an open space on the block diagram to display the *Functions* palette. The VIs and functions located on the *Functions* palette depend on the palette view currently selected. The VIs and functions are located on subpalettes based on the types of VIs and functions.

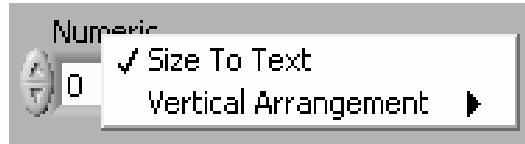


**Figure 2.6** Functions palette.

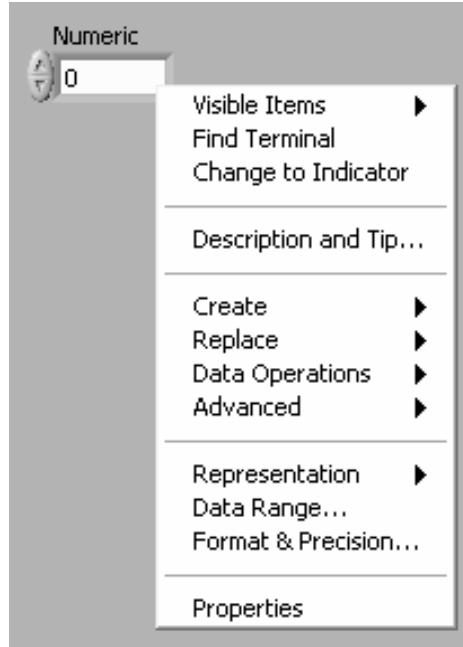
## 2.8 SHORTCUT MENUS

All LabVIEW objects have associated shortcut menus. To access the shortcut menu right-click the object and change the look or behavior of front panel and block diagram objects. The most often-used menu is the object shortcut menu. All LabVIEW objects and empty space on the front panel and block diagram have associated shortcut menus.

Use the shortcut menu items to change the look or behavior of front panel and block diagram objects. To access the shortcut menu, right-click the object, front panel or block diagram. To access the shortcut menu, right-click the object. Right-click the label as shown in Figure 2.7 to access its shortcut menu for front panel objects. Right-click the digital display to access its shortcut menu as shown in Figure 2.8. The shortcut menu for a meter is shown in Figure 2.9.

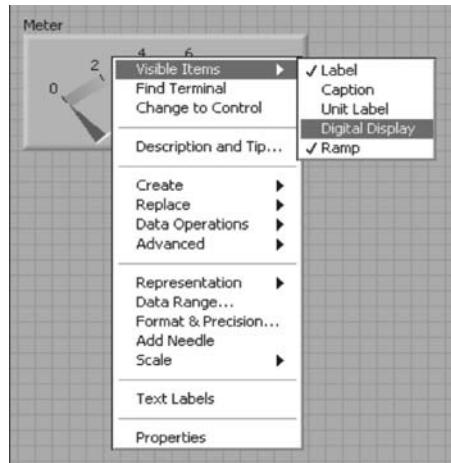


**Figure 2.7** Label shortcut menu.



**Figure 2.8** Display shortcut menu.

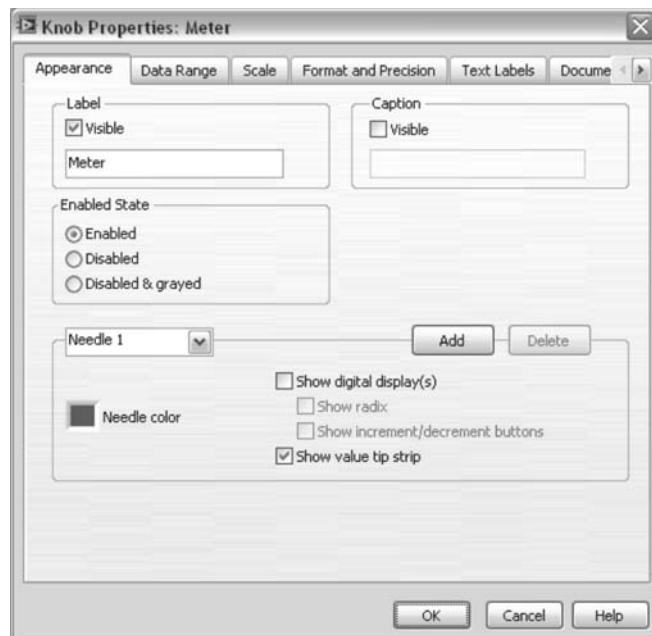
You can use the shortcut menu to create constants, controls and indicators. Right-click a function terminal and select *Create»Constant*, *Create»Control*, or *Create»Indicator* from the shortcut menu to create. To replace nodes, right-click the node and select *Replace* from the shortcut menu.



**Figure 2.9** Meter shortcut menu.

## 2.9 PROPERTY DIALOG BOXES

Front panel objects also have property dialog boxes that you can use to change the look or behavior of front panel objects. Right-click the front panels object and select *Properties* from the shortcut menu to access the property dialog box for an object. Figure 2.10 shows the property dialog box for a meter. The options available on the property dialog box for an object are similar to the options available on the shortcut menu for that object.

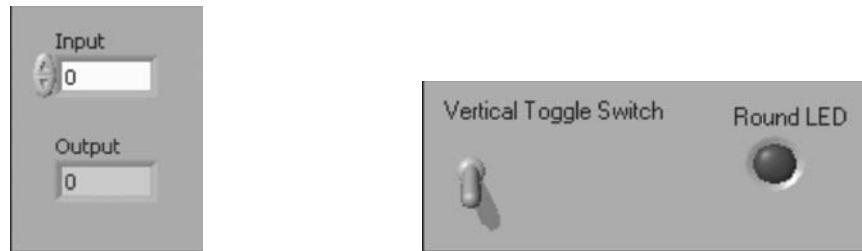


**Figure 2.10** Property dialog box for a meter.

## 2.10 FRONT PANEL CONTROLS AND INDICATORS

You can build the front panel with controls and indicators, which are the interactive input and output terminals of the VI, respectively. Controls are knobs, push buttons, dials and other input devices. Indicators are graphs, LEDs and other displays. Controls simulate instrument input devices and supply data to the block diagram of the VI. Indicators simulate instrument output devices and display data the block diagram acquires or generates.

Every control or indicator has a data type associated with it. They are numeric data type, Boolean data type and string data type as shown in Figure 2.11. The numeric data type shown in Figure 2.11(a) can be of various types such as integer or real. The two most commonly used numeric objects are the numeric control and the numeric indicator. The Boolean data type represents data that only has two parts, such as TRUE and FALSE or ON and OFF as in Figure 2.11(b). Use Boolean controls and indicators to enter and display Boolean (True or False) values. Boolean objects simulate switches, push buttons, and LEDs. Figure 2.11(c) shows the string data type which is a sequence of ASCII characters. Use string controls to receive text from the user such as a password or user name and indicators to display text to the user.



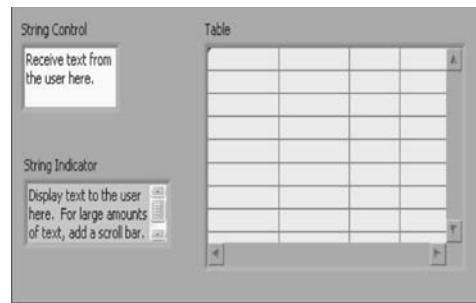
(a) Numeric controls and indicators

*Input:* Numeric control with increment decrement button  
*Output:* Numeric indicator



(b) Boolean controls and indicators

*Vertical toggle switch:* Boolean controls  
*Round LED:* Boolean indicators



(c) String controls and indicators

String controls to receive text from the user such as a password or user name  
 String indicators to display text to the user

**Figure 2.11** Numeric, Boolean and string data type.

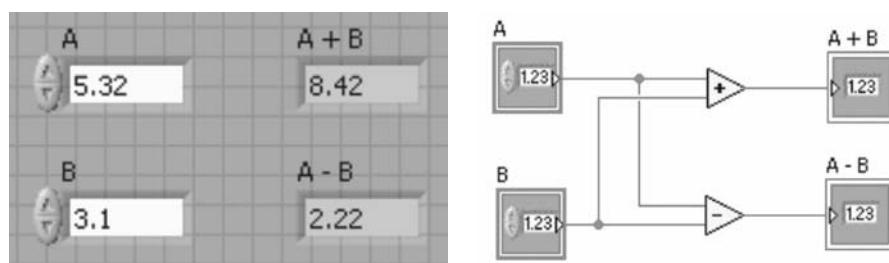
To keep an object proportional to its original size as you resize it, press the <Shift> key while you drag the resizing handles or circles. To resize an object as you place it on the front panel, press the <Ctrl> key while you click to place the object and drag the resizing handles or circles.

## 2.11 BLOCK DIAGRAM

Block diagram objects include terminals, subVIs, functions, constants, structures and wires to transfer data among other block diagram objects.

### 2.11.1 Terminals

Front panel objects appear as terminals on the block diagram. Figure 2.12 shows the front panel where two numeric values A and B need to be added and subtracted, and the block diagram for the corresponding front panel. Terminals are entry and exit ports that exchange information between the front panel and block diagram. Terminals are analogous to parameters and constants in text-based programming languages. Types of terminals include control or indicator terminals and node terminals. Control and indicator terminals belong to front panel controls and indicators. Data you enter into the front panel controls (A and B in the figure) enter the block diagram through the control terminals. The data then enter the Add and Subtract functions. When the Add and Subtract functions complete their calculations, they produce new data values. The data values flow to the indicator terminals, where they update the front panel indicators.



**Figure 2.12** Front panel and its corresponding block diagram.

The terminals represent the data type of the control or indicator. You can configure front panel controls or indicators to appear as icon or data type terminals on the block diagram. By default, front panel objects appear as icon terminals. For example, a numeric icon terminal, shown in Figure 2.13, represents a numeric control on the front panel. A DBL terminal represents a double-precision, floating-point numeric control. To display a terminal as a data type on the block diagram, right-click the terminal and select *View As Icon* from the shortcut menu.



**Figure 2.13** Default icon terminals can be viewed as data type terminal.

### 2.11.2 Nodes

Nodes are objects on the block diagram that have inputs and/or outputs and perform operations when a VI runs. They are analogous to statements, operators, functions, and subroutines in text-based programming languages. Nodes can be functions, subVIs or structures. Structures are process control elements, such as Case structures, For loops, or While loops. The Add and Subtract functions in the previous figure are function nodes.

### 2.11.3 Functions

Functions are the fundamental operating elements of LabVIEW. Functions do not have front panels or block diagrams but do have connector panes. Double-clicking a function only selects the function. A function has a pale yellow background on its icon.

### 2.11.4 SubVIs

SubVIs are VIs that you build to use inside of another VI or that you access on the *Functions* palette. Any VI has the potential to be used as a subVI. When you double-click a subVI on the block diagram, its front panel and block diagram appear. The front panel includes controls and indicators. The block diagram includes wires, front panel icons, functions, possibly subVIs and other LabVIEW objects. The upper-right corner of the front panel and block diagram displays the icon for the VI. This is the icon that appears when you place the VI on a block diagram as a subVI. SubVIs also can be Express VIs. Express VIs are nodes that require minimal wiring because you configure them with dialog boxes. Use Express VIs for common measurement tasks. You can save the configuration of an Express VI as a subVI. Refer to the *Express VIs* topic of the *LabVIEW Help* for more information about creating a subVI from an Express VI configuration.

### 2.11.5 Express VIs and VIs

Express VIs are interactive VIs with configurable dialog page but Standard VIs are modularized VIs customized by wiring. LabVIEW uses colored icons to distinguish between Express VIs, VIs and functions on the block diagram. By default, icons for Express VIs appear on the block diagram as expandable nodes with icons surrounded by a blue field. Icons for VIs have white backgrounds, and icons for functions have pale yellow backgrounds. By default, most functions and VIs on the block diagram appear as icons that are not expandable, unlike Express VIs.

### 2.11.6 Wires

You can transfer data among block diagram objects through wires. Each wire has a single data source, but you can wire it to many VIs and functions that read the data. Wires are different colors, styles and thicknesses, depending on their data types as shown in Table 2.5. A broken wire appears as a dashed black line with a red X in the middle, as shown at left. Broken wires occur for a variety of reasons, such as when you try to wire two objects with incompatible data types. You must connect the wires to inputs and outputs that are compatible with the data that is transferred with the wire. You cannot wire an array output to a numeric input. In addition, the direction of the wires must be correct. You must connect the wires to only one input and at least one output. Also you

cannot wire two indicators together. The components that determine wiring compatibility include the data type of the control and/or the indicator and the data type of the terminal. Press <Ctrl>-B to delete all broken wires or right-click and select *Clean Up Wire* to reroute the wire.

**TABLE 2.5** Common wire types

Data type	Scalar	1D Array	2D Array
DBL Numeric (Floating Point)	—	—	—
Integer Numeric	—	—	—
Boolean	- - - - -	- - - - -	- - - - -
String	~~~~~	~~~~~	~~~~~
Dynamic	-----	-----	-----

LabVIEW automatically wires objects as you place them on the block diagram. LabVIEW connects the terminals that best match and leave terminals that do not match unconnected. As you move a selected object close to other objects on the block diagram, LabVIEW draws temporary wires to show you valid connections. When you release the mouse button to place the object on the block diagram, LabVIEW automatically connects the wires. Toggle automatic wiring by pressing the spacebar while you move an object using the Positioning tool. You can adjust the automatic wiring settings by selecting *Tools»Options* and selecting *Block Diagram* from the top pull-down menu.

## 2.12 DATA TYPES

Data types indicate what objects, inputs and outputs you can wire together. For example, if a switch has a green border, you can wire a switch to any input with a green label on an Express VI. If a knob has an orange border, you can wire a knob to any input with an orange label. However, you cannot wire an orange knob to an input with a green label. Notice the wires are the same color as the terminal. The dynamic data type stores the information generated or acquired by an Express VI. The dynamic data type appears as a dark blue terminal. Most Express VIs accept and/or return the dynamic data type. You can wire the dynamic data type to any indicator or input that accepts numeric, waveform or Boolean data. Wire the dynamic data type to an indicator that can best present the data. Indicators include a graph, chart or numeric indicator.

## 2.13 DATA FLOW PROGRAM

LabVIEW follows a dataflow model for running VIs. A block diagram node executes when all its inputs are available. When a node completes execution, it supplies data to its output terminals and passes the output data to the next node in the dataflow path.

Visual Basic, C++, JAVA, and most other text-based programming languages follow a control flow model of program execution. In control flow, the sequential order of program elements determines the execution order of a program. For a data flow programming, consider a block diagram shown in Figure 2.14 that adds two numbers and then subtracts 50.00 from the result of the addition. In this case, the block diagram executes from left to right, not because the objects are placed in that order, but because the Subtract function cannot execute until the Add function finishes executing and passes the data to the Subtract function. A node executes only when data are available at all of its input terminals, and it supplies data to its output terminals only when it finishes execution.

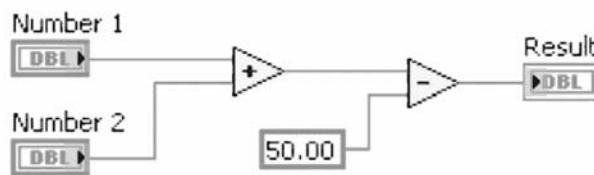


Figure 2.14 Data flow program.

## 2.14 LabVIEW DOCUMENTATION RESOURCES

The programmer can use the *Context Help* window, the *LabVIEW Help* and the *NI Example Finder* to help build and edit VIs. The *Context Help* window displays basic information about LabVIEW objects when you move the cursor over each object. To toggle display of the Context Help window, select *Help»Show Context Help*, press <Ctrl-H> or click the *Show Context Help Window* button on the toolbar.

The *LabVIEW Help* contains detailed descriptions of most palettes, menus, tools, VIs and functions. It *LabVIEW Help* also includes step-by-step instructions for using LabVIEW features. Access *LabVIEW Help* by the following methods:

- Click the *More Help* button in the *Context Help* window
- Select *Help»Search the LabVIEW Help*
- Use the *Click here* for more help link in the *Context Help* window
- Right-click an object and select *Help*

The *New* dialog box contains many LabVIEW template VIs that you can use to start building VIs. However, these template VIs are only a subset of the hundreds of example VIs included with LabVIEW. You can modify any example VI to fit an application, or you can copy and paste from an example into a VI that you create. In addition to the example VIs that ship with LabVIEW, you also can access hundreds of example VIs on the NI Developer Zone at [ni.com/zone](http://ni.com/zone). To search all examples using LabVIEW VIs, use the *NI Example Finder*. The *NI Example Finder* is the gateway to all installed examples and the examples located on the *NI Developer Zone*. To launch the *NI Example Finder*, select *Help»Find Examples* from the front panel or block diagram menu bar. You also can launch the *NI Example Finder* by clicking the arrow on the *Open* button on the LabVIEW dialog box and selecting *Examples* from the shortcut menu.

## 2.15 KEYBOARD SHORTCUTS

Frequently used menu options have equivalent keyboard shortcuts:

- <Ctrl-S> Save a VI
- <Ctrl-R> Run a VI.
- <Ctrl-E> Toggle between the front panel and block diagram.
- <Ctrl-H> Activate *Context Help* window.
- <Ctrl-B> Remove all broken wires.
- <Ctrl-F> Find object
- <shift>-right-click to access *Tools Palette*

Press the <Ctrl> key while using the positioning tool to click and drag a selection to *duplicate* an object.

---

## SUMMARY

---

- Virtual instruments (VIs) have three main parts — the front panel, the block diagram, and the icon and connector pane.
- The front panel is the user interface of a LabVIEW program and specifies the inputs and displays the outputs of the VI.
- Place controls (inputs) and indicators (outputs) in the front panel window.
- Control terminals have thicker borders than indicator terminals.
- All front panel objects have property pages and shortcut menus.
- The block diagram contains the executable graphical source code composed of nodes, terminals, wires, Express VIs, standard VIs and functions on the block diagram to create measurement code.
- Use the *Wiring* tool to connect diagram objects.
- To change a control to an indicator or to change an indicator to a control, right-click the object and select *Change to Indicator* or *Change to Control* from the shortcut menu.
- The broken *Run* button appears on the toolbar to indicate the VI is nonexecutable. Click the broken *Run* button to display the *Error list* window, which lists all the errors.
- Various debugging tools and options such as setting probes and breakpoints, execution highlighting, and single stepping are available.
- The three floating palettes are *Tools Palette*, *Controls Palette* (only when Front Panel Window is active) and *Functions Palette* (only when Block Diagram Window is active).
- Use the *Tools* palette to create, modify and debug VIs. Press the <Shift> key and right-click to display a temporary version of the *Tools* palette at the location of the cursor.
- Use the *Controls* palette to place controls and indicators on the front panel. Right-click an open space on the front panel to display the *Controls* palette.
- Use the *Functions* palette to place VIs and functions on the block diagram. Right-click an open space on the block diagram to display the *Functions* palette.

- Use the *Search* button on the *Controls* and *Functions* palettes to search for controls, VIs and functions.
- All LabVIEW objects and empty space on the front panel and block diagram have associated shortcut menus, which you access by right-clicking an object, the front panel or the block diagram.
- Use execution highlighting, single-stepping, probes and breakpoints to debug VIs by animating the flow of data through the block diagram.
- Use the *Help* menu to display the *Context Help* window and the *LabVIEW Help*, which describes most palettes, menus, tools, VIs, functions and features.

## MISCELLANEOUS SOLVED PROBLEMS

### Problem 2.1 Check controls and indicators:

- Numeric Control and Indicator
- Boolean Control and Indicator
- String Control and Indicator
- Dial connected to Gauge and Meter
- Thermometer to Numeric Indicator
- Tank to Numeric Indicator
- Vertical Slide Control to Horizontal Slide Control

**Solution** The front panel with controls and indicators and the respective block diagrams are shown in Figures P2.1(a) and P2.1(b).

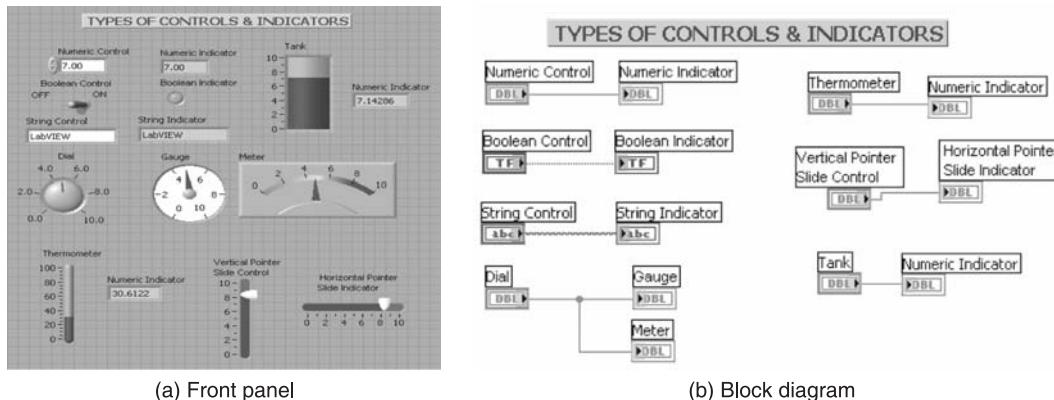


Figure P2.1

### Problem 2.2 Add, multiply, subtract and divide two numeric inputs.

**Solution** The front panel has controls A and B and four indicators as shown in Figure P2.2(a). The block diagram has the respective terminals and functions as shown in Figure P2.2(b).

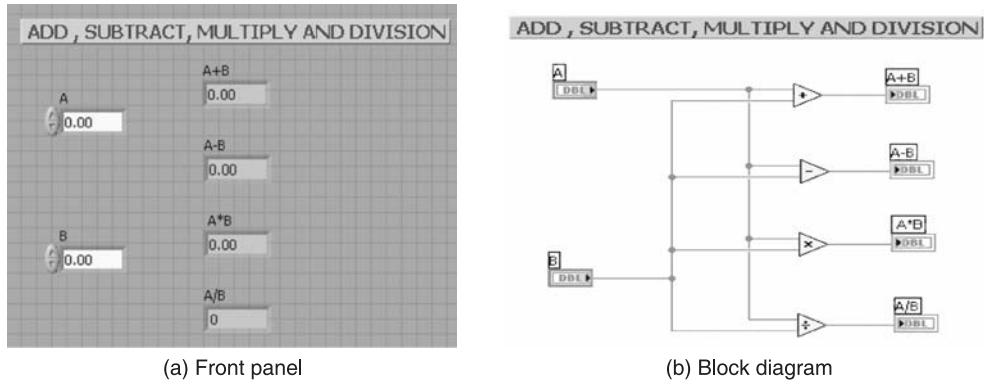


Figure P2.2

**Problem 2.3** Add and multiply more than two numeric inputs.

**Solution** The front panel has three numeric inputs while the block diagram contains compound arithmetic functions as shown in Figures P2.3(a) and P2.3(b).

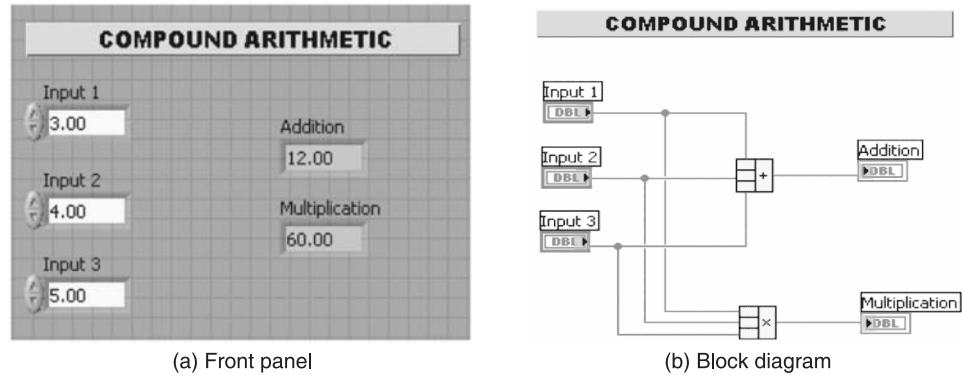


Figure P2.3

**Problem 2.4** Divide two numbers and find the remainder and quotient.

**Solution** The front panel and block diagram for finding the remainder and quotient are shown in Figures P2.4(a) and P2.4(b).

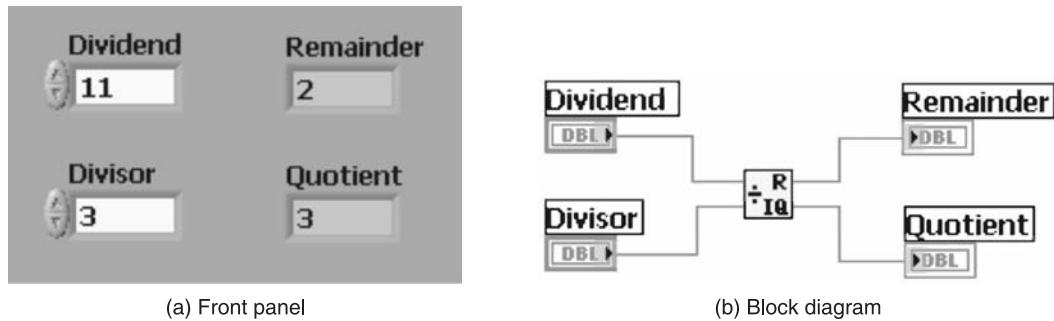


Figure P2.4

**Problem 2.5** Compute the expressions  $Y = (A*B*C) + (D*E)$  and  $Y = mx + c$ .

**Solution** The front panel and block diagram for solving two equations are shown in Figures P2.5(a) and P2.5(b).

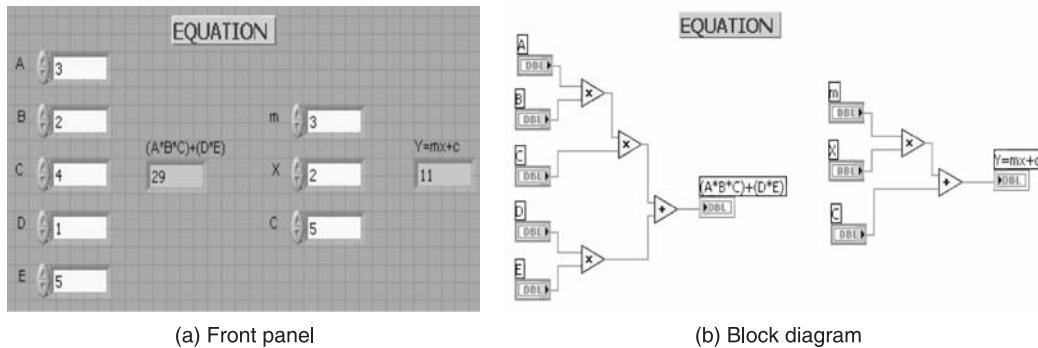


Figure P2.5

**Problem 2.6** Compute the equations  $(X1+2)*\log(X1)$  using functions, Expression node and Express Formula for the given inputs X1.

**Solution** The front panel and block diagram for solving an equation using three different ways are shown in Figures P2.6(a) and P2.6(b).

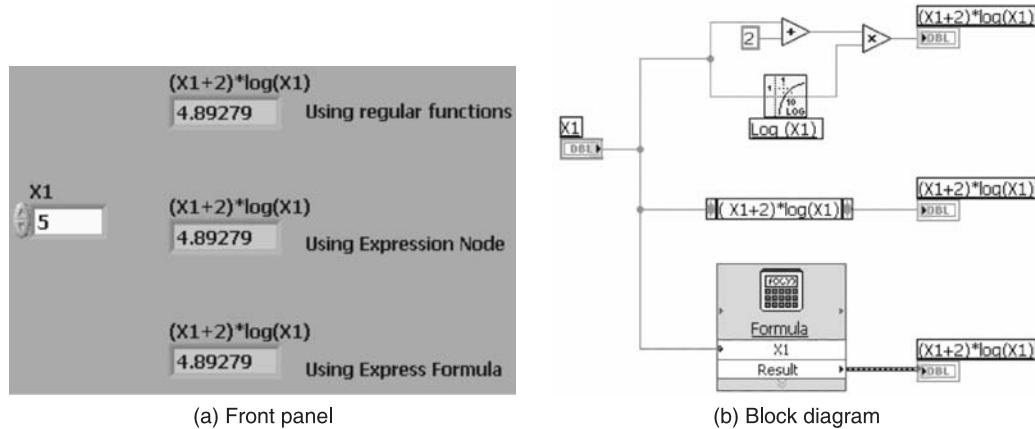


Figure P2.6

**Problem 2.7** Convert Celsius to Fahrenheit.

**Solution** The front panel and block diagram for solving the equation  $F = (1.8C + 32)$  are shown in Figures P2.7(a) and P2.7(b).

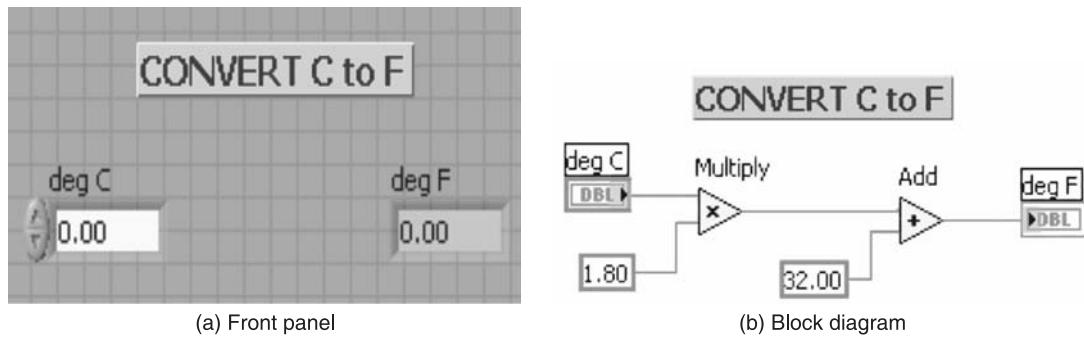


Figure P2.7

**Problem 2.8** Find whether the given number is odd or even.

**Solution** Click continuous run and increment the number to see the even and odd LED glow alternatively. The front panel and block diagram are shown in Figures P2.8(a) and P2.8(b).

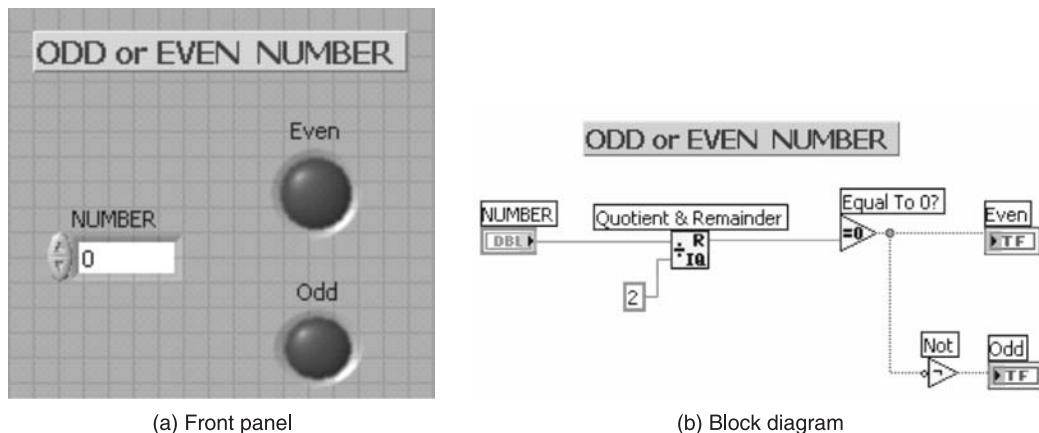


Figure P2.8

**Problem 2.9** Convert radians to degrees and degrees to radians.

**Solution** The front panel and block diagram are shown in Figures P2.9(a) and P2.9(b) where Degrees = Radians ( $180/\pi$ ).

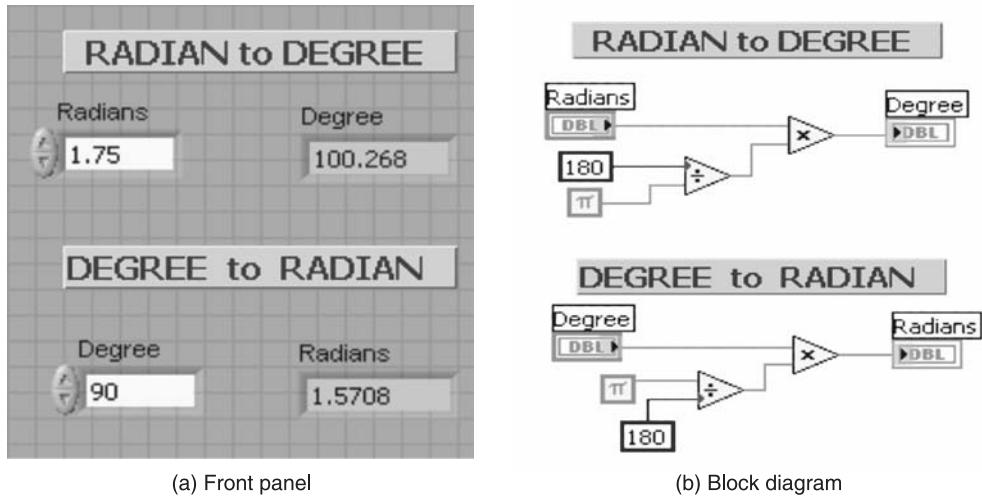


Figure P2.9

**Problem 2.10** Find trigonometric values for the given angle. In LabVIEW computation of trigonometric values such as sine, cosine and tangent takes the input in radians. Hence, angles in degrees are to be converted to radians before calculation.

**Solution** The angle is provided as control and the corresponding radians, sin, cos and tan values are displayed. The front panel and block diagram are shown in Figures P2.10(a) and P2.10(b).

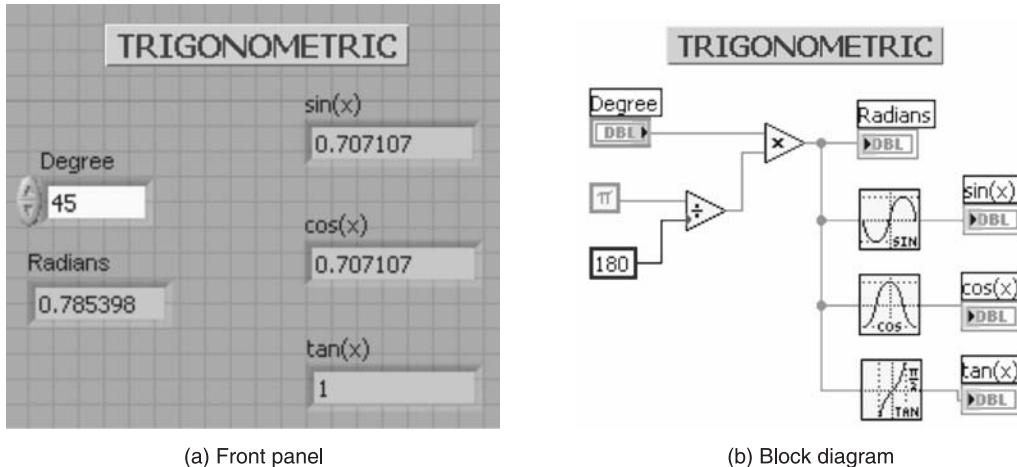


Figure P2.10

**Problem 2.11** Perform various Boolean Operations (AND, OR, NAND, NOR, XOR).

**Solution** The front panel and block diagram are shown in Figures P2.11(a) and P2.11(b) and the truth table for Boolean operations can be checked.

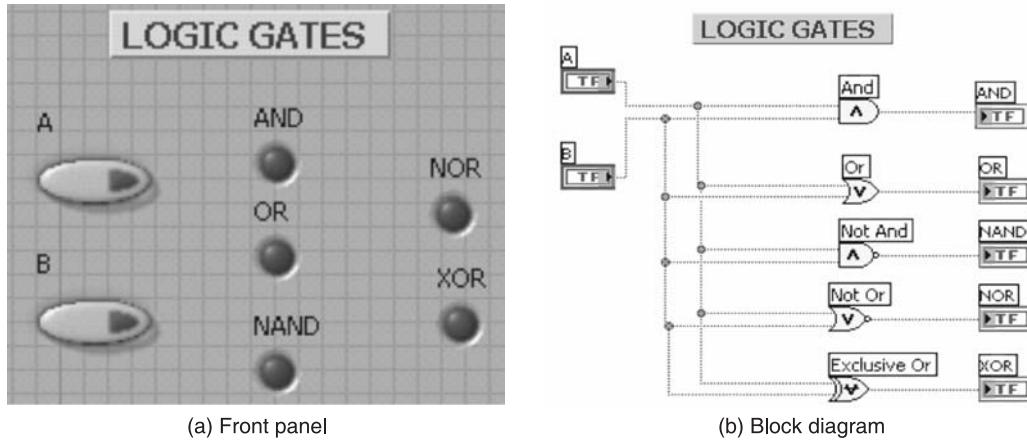


Figure P2.11

**Problem 2.12** Convert a binary number to a decimal number.

**Solution** The front panel and block diagram are shown in Figures P2.12(a) and P2.12(b) to convert a binary to a digital number.

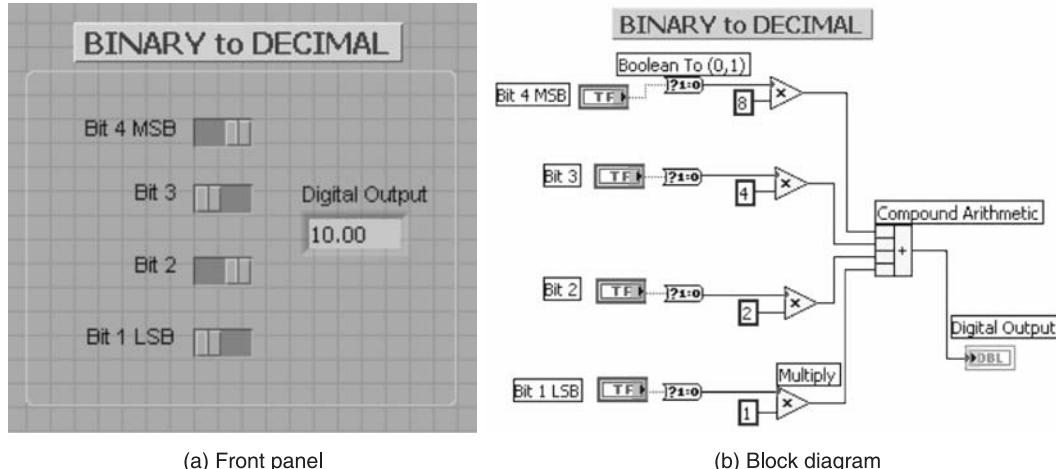


Figure P2.12

**Problem 2.13** Split an input string into two outputs with reference to a separating character. Find the length of the input string and reverse the string.

**Solution** Any input string can be provided. In this problem “LABVIEW BASICS” is given and the front panel and block diagram are shown in Figures P2.13(a) and P2.13(b).

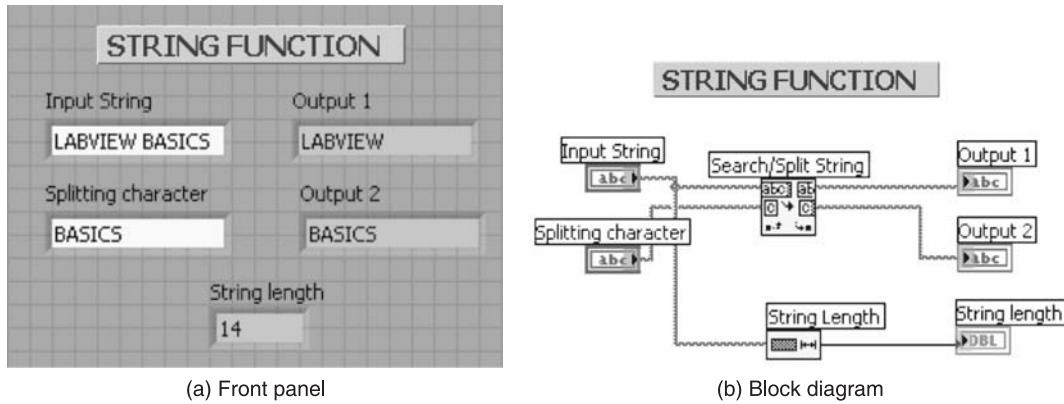


Figure P2.13

**Problem 2.14** Add two binary bits and find the sum and carry (half adder).

**Solution** The front panel and block diagram are shown in Figures P2.14(a) and P2.14(b) and the half adder can be checked.

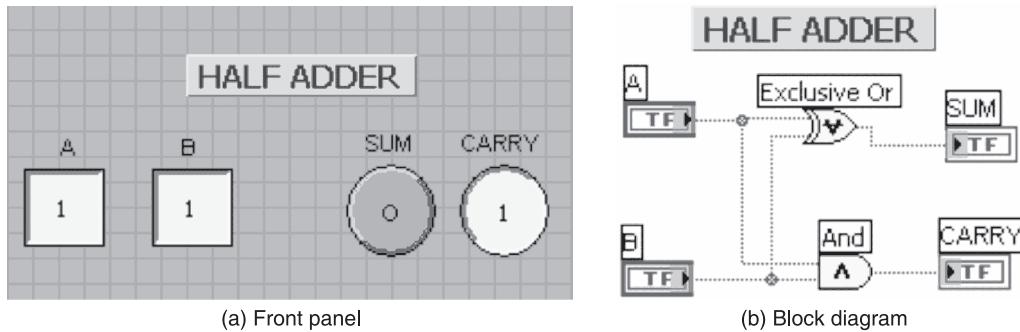


Figure P2.14

## REVIEW QUESTIONS

1. Why is LabVIEW a graphical programming language?
2. How is LabVIEW different from text-based programming languages?
3. Explain the typical features and advantages of LabVIEW.
4. Explain the salient features in LabVIEW software environment.
5. What are the three main components of a virtual instrument (VI)?
6. What are the three palettes used in programming?
7. How can you create, modify and debug VIs using floating tools palette?
8. Explain the features in the front panel toolbar used in programming.
9. LabVIEW follows a data flow model for running VIs. Explain with an example.

10. Explain the importance of the toolbar buttons that appear on the block diagram.
11. Explain the debugging techniques in LabVIEW programming.
12. What are Express VIs and standard VIs?
13. With the help of an example explain the block diagram objects.
14. Explain data types and the importance of their colors in LabVIEW programs with examples.

### **EXERCISES**

---

1. The length and breadth of a rectangle and the radius of a circle are inputs. Build a VI to calculate the area and perimeter of the rectangle, and the area and circumference of the circle.
2. Divide two numbers and glow an LED if the result of the division is infinity (i.e. the divisor is zero).
3. Create NOT, AND and OR gates using NAND gate and verify their truth table.
4. Perform OR and AND operations of an array of Boolean inputs.
5. Perform AND, OR and XOR operations using more than two Boolean inputs (individual inputs, not an array of inputs).
6. The salary of X is input. His dearness allowance is 40% of basic salary and house rent allowance is 20% of basic salary. Build a VI to calculate his gross salary.
7. Check whether the given string input is empty, a space, printable ACSII character, hexadecimal, digit and octal number.
8. Form a complex number using two inputs x and y. For that complex number, find the complex conjugate and polar components.
9. Check whether the given number is in the limit (specify upper and lower limits).
10. The population of a town is 80,000 and the percentage of men is 52. The percentage of total literacy is 48. If the total percentage of literate men is 35 of the total population, build a VI to find the total number of illiterate men and women.
11. Find the hexadecimal value of the given input string and display the hexadecimal values in an array.
12. Find the equivalent gray code for a given BCD.
13. Find the equivalent BCD of an input binary value.
14. Design a decoder for two binary inputs.
15. Find logarithmic values for inputs  $x$  and  $y$  ( $\log(x)$  and  $\log(y)$ ). Add the log values and find  $10^{(\log(x)+\log(y))}$ . Verify that the result is equal to the product of  $x$  and  $y$ .
16. Design a  $4 \times 1$  multiplexer with enable and select options.
17. Compute the equations  $(X1 + 2)*3$  and  $5 + X2*\log(X2)$  using functions, Expression node and Express Formula for the given inputs  $X1$  and  $X2$ .
18. For the two Boolean inputs use Implies Function and verify the results.
19. From the given two numeric inputs, find the maximum value and minimum value.
20. Change the dimension of a LED using property node and also change the position of the LED.

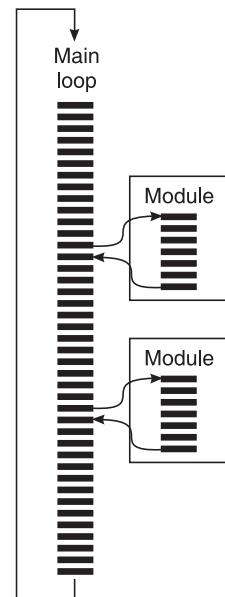


# MODULAR PROGRAMMING

## 3.1 INTRODUCTION

Modular programming refers to the idea that programs are easier to read, to write, to debug, and to maintain if they are divided into smaller subprograms. The benefits of modular programming are several. First of all, it makes our programs easier to write because individual components can be independently written and tested. Second, it makes the “main” part of the code easier to read since long code sections are replaced with simple functions (whose internal code is hidden in another file). This also makes our programs simpler to modify. Third, individual components can be reused in other programs. For example, suppose you write a program that accepts data from the keyboard and calculates the average, standard deviation and so on. If the part of the program that calculates the standard deviation is contained inside a separate function, you could reuse that function in another program that needed to calculate the standard deviation.

Even though the simple program structure works well for simple example, it is counter-productive for longer programs, leading to lack of clarity and slowing code

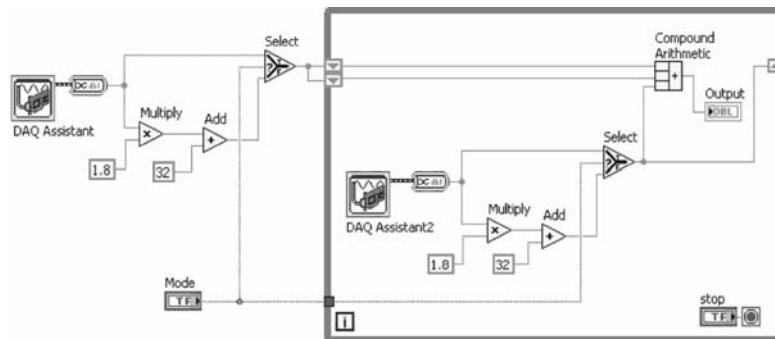


**Figure 3.1** Modular programming.

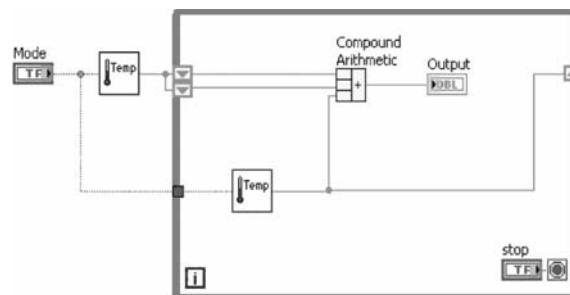
maintenance and modification. Each well-defined task in a program should be placed in its own program unit. Here you will execute the tasks in program units called *subroutines*. The main program simply acts as an outline or driver, triggering execution of the program units that accomplish the tasks. Figure 3.1 shows the modular programming approach. This chapter describes modular programming approach in LabVIEW software.

### 3.2 MODULAR PROGRAMMING IN LabVIEW

The power of LabVIEW lies in the hierarchical nature of the VI. After you create a VI, you can use it on the block diagram of another VI. There is no limit on the number of layers in the hierarchy. Modular programming helps manage changes and debug the block diagram quickly. Modularity defines the degree to which your VI is composed of discrete components such that a change to one component has minimal impact on other components. These components are called *modules* or *subVIs*. Modularity increases the readability and reusability of your VIs. As you create VIs, you might find that you perform a certain operation frequently. Consider using subVIs or loops to perform that operation repetitively. In Figure 3.2 the block diagram contains two identical operations to convert temperature in Celsius to Fahrenheit inside and outside the loop. You can create a subVI that performs that operation and call the subVI twice. The example in Figure 3.3 calls this temperature VI as a subVI twice on its block diagram and functions the same as the previous block diagram. You also can reuse the subVI in other VIs.



**Figure 3.2** Block diagram with two identical operations.



**Figure 3.3** Temperature VI as a subVI called twice on its block diagram.

A VI within another VI is called a subVI. A subVI corresponds to a subroutine in text-based programming languages. You can reuse a subVI in other VIs. If you use a VI as a subVI, the icon identifies the subVI when it is called from the block diagram of another VI. Windows Explorer also uses the icon to identify the .vi file if you enable this feature. The steps to create a subVI is, first build a VI front panel and block diagram, next build the icon and the connector pane and then you can use the VI as a subVI.

### 3.3 BUILD A VI FRONT PANEL AND BLOCK DIAGRAM

The first step is to create the front panel and block diagram of a VI as shown in Figure 3.4. Consider a VI slope (m).vi to find the slope m given the coordinates  $y_1$ ,  $y_2$ ,  $x_1$ ,  $x_2$  of a line. The slope  $m = (y_2 - y_1)/(x_2 - x_1)$ .

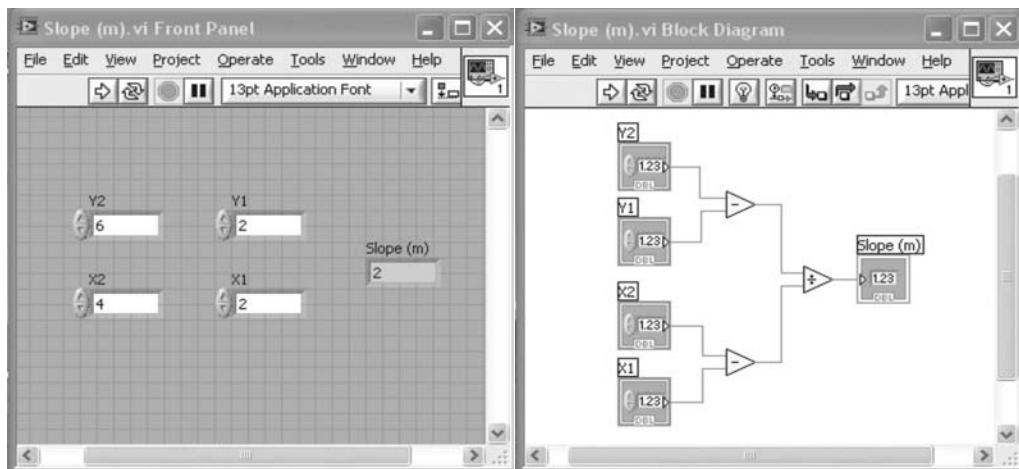


Figure 3.4 VI to find the slope of a line.

### 3.4 ICON AND CONNECTOR PANE

After you build a VI, build the icon and the connector pane so that you can use the VI as a subVI. The icon and connector pane correspond to the function prototype in text-based programming languages. Every VI displays an icon in the upper-right corner of the front panel and block diagram windows. An icon is a graphical representation of a VI. If you use a VI as a subVI, then the icon identifies the subVI on the block diagram of the VI.

The default icon can be changed to a custom icon which contains text, images or a combination of both. Figure 3.5 shows the custom icon and connector pane created for the VI to find the slope of a given line. The icon for the slope VI contains text and images while the connector has four inputs ( $y_1$ ,  $y_2$ ,  $x_1$ ,  $x_2$ ) and one output ( $m$ ). To create a subVI, you need to build an icon and a connector pane.

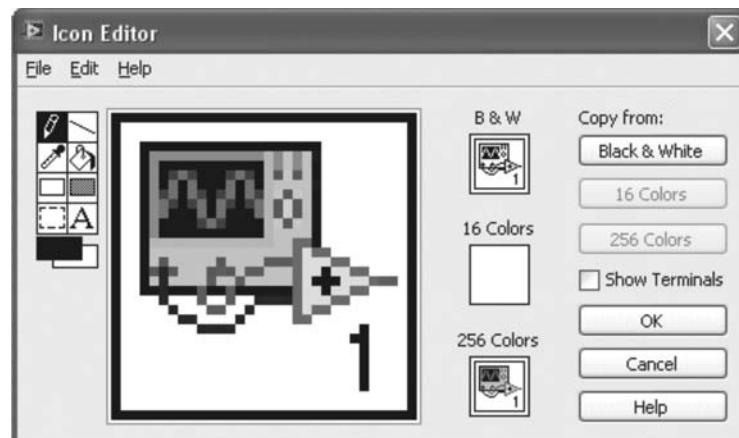


**Figure 3.5** Icon and connector pane created for the slope VI.

### 3.5 CREATING AN ICON

The default icon of a VI contains a number that indicates how many new VIs you have opened since launching LabVIEW. You can create custom icons to replace the default icon by completing the following steps:

1. Right-clicking the icon in the upper-right corner of the front panel or block diagram and select **Edit Icon** from the shortcut menu to display the *Icon Editor* dialog box as shown in Figure 3.6.

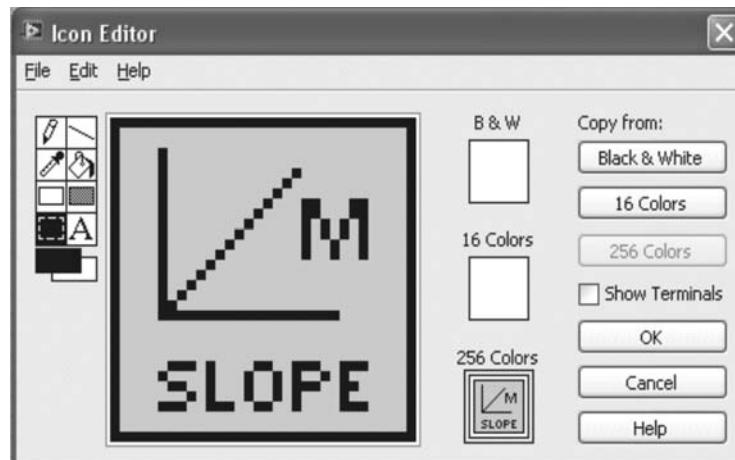


**Figure 3.6** Default Icon Editor dialog box.

2. Double-click the hatched box which will select the entire icon. Delete the selected portion.
3. Double-click the rectangular box which will create a border for the icon.
4. Use the line/pencil tool to draw.
5. Double-click the *Edit Text* tool 'A' to edit the required text.
6. Choose the *Text Tool Font* to edit font, font size, color and alignment of the text.
7. Use the *Select Color* tool to choose the background color of the icon.
8. Use the *Fill With Color* tool to change the background color of the icon.
9. Click *OK* to save the icon.

Using the tools on the icon editor, Figure 3.7 shows the custom Icon created for the slope VI. Use the **Edit** menu to cut, copy and paste images from and to the icon. When you select a portion of the icon and paste an image, LabVIEW resizes the image to fit into the selection area. You also can drag a graphic from anywhere in the file system and place it in the upper-right corner of the

front panel. LabVIEW converts the graphic to a 32×32 pixel icon. Depending on the type of monitor you use, you can design a separate icon for monochrome, 16-color and 256-color mode. Use the *Copy from* option on the right side of the *Icon Editor* dialog box to copy from a color icon to a black-and-white icon and vice versa. After you select a *Copy from* option, click the *OK* button to complete the change. Figure 3.7 shows the custom icon created for the VI which finds the slope of a line.



**Figure 3.7** Custom Icon Editor created for the slope VI.

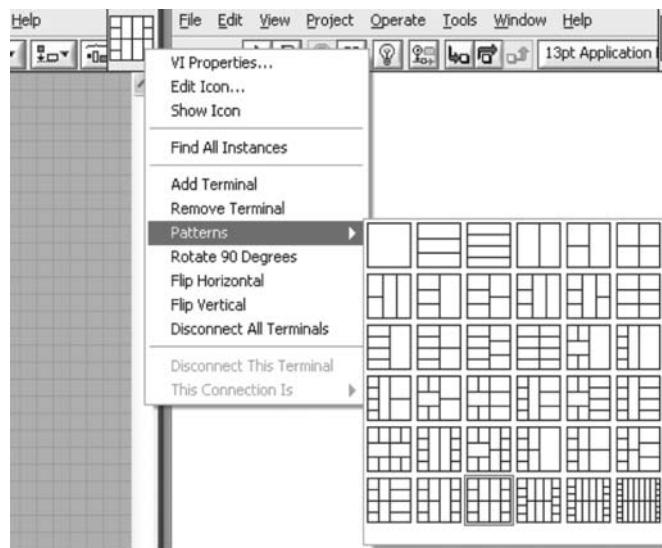
### 3.6 BUILDING A CONNECTOR PANE

The connector pane is a set of terminals that correspond to the controls and indicators of that VI, similar to the parameter list of a function call in text-based programming languages. The connector pane defines the inputs and outputs you can wire to the VI so that you can use it as a subVI. A connector pane receives data at its input terminals and passes the data to the block diagram code through the front panel controls or receives the results at its output terminals from the front panel indicators.

To define a connector pane, right-click the icon in the upper-right corner of the front panel and select *Show Connector* from the shortcut menu to display the connector pane. The connector pane appears in place of the icon. When you view the connector pane for the first time, you see a default connector pattern. You can select an appropriate pattern by right-clicking the connector pane and selecting *Patterns* from the shortcut menu as shown in Figure 3.8. After you select a connector pane pattern, you can customize it to suit the VI by adding, removing or rotating the terminals.

To add a terminal to the pattern, place the cursor where you want to add the terminal, right-click, and select *Add Terminal* from the shortcut menu. To remove an existing terminal from the pattern, right-click the terminal and select *Remove Terminal* from the shortcut menu. To change the spatial arrangement of the connector pane patterns, right-click the connector pane and select *Flip Horizontal*, *Flip Vertical*, or *Rotate 90 Degrees* from the shortcut menu. Assign a front panel

control or indicator to each of the connector pane terminals. If you placed the VI as a subVI on another block diagram, you must relink the subVI to the VI whose connector pane you changed by right-clicking the subVI and selecting *Relink To SubVI* from the shortcut menu. Otherwise, the VI containing the subVI is broken and will not run.



**Figure 3.8** Selecting different connector patterns.

Each rectangle on the connector pane represents a terminal. Use the rectangles to assign inputs and outputs. The default connector pane pattern is  $4 \times 2 \times 2 \times 4$ . If you anticipate changes to the VI that would require a new input or output, keep the default connector pane pattern to leave extra terminals unassigned. You can assign up to 28 terminals to a connector pane. If our front panel contains more than 28 controls and indicators that you want to use programmatically, group some of them into a cluster and assign the cluster to a terminal on the connector pane. Assigning more than 16 terminals to a VI can reduce readability and usability.

### 3.6.1 Assigning Terminals to Controls and Indicators

After you select a pattern to use for the connector pane, you must assign a front panel control or indicator to each of the connector pane terminals. To link controls and indicators to the connector pane, place inputs on the left and outputs on the right to prevent complicated or confusing wiring patterns.

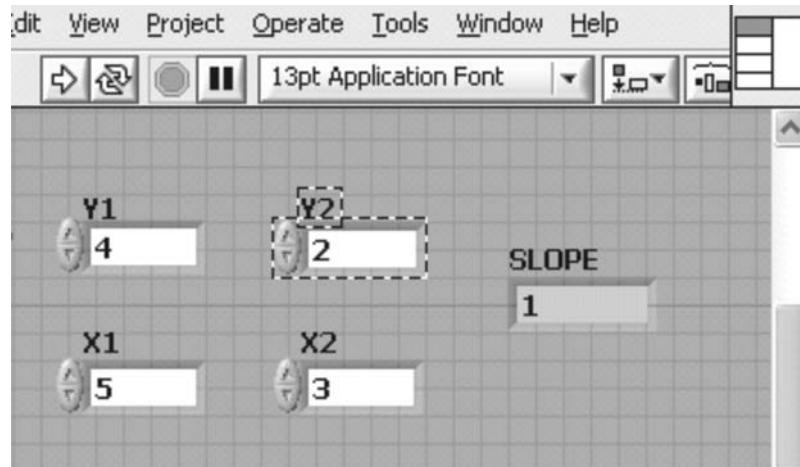
Complete the following steps to assign terminals to controls and indicators in a connector pane.

**Step 1:** Ensure that you have selected a pattern sufficient for the number of controls and indicators you want to assign to the connector pane.

**Step 2:** Right-click the icon in the upper-right corner of the front panel and select *Show Connector* from the shortcut menu to display the connector pane. The connector pane appears in place of the icon.

**Step 3:** Click a terminal of the connector pane. The tool automatically changes to the wiring tool and the terminal turns black.

**Step 4:** Click the front panel control or indicator you want to assign to the terminal. A marquee highlights the object as shown in Figure 3.9.



**Figure 3.9** Assigning terminals to controls and indicators of the slope VI.

**Step 5:** Click an open space of the front panel. The marquee disappears, and the terminal changes to the data type color of the control to indicate that you connected the terminal. If the connector pane terminal turns white, a connection was not made. Repeat steps 1 through 3 until the connector pane terminal changes to the proper data type color.

**Step 6:** Repeat steps 3 through 5 for each control and indicator you want to assign to a terminal. If you need to change the control or indicator assigned to a terminal, you must first delete the connection and repeat steps 3 through 5 to assign another control or indicator to the terminal.

**Step 7:** If necessary, confirm each terminal connection. You can specify which terminals are required, recommended and optional. You can connect only one control or indicator to a terminal.

### 3.6.2 Confirming Terminal Connections

To confirm which control or indicator is assigned to a connector pane terminal, click the terminal in the connector pane. A marquee highlights the assigned object. You also can use the wiring tool to click the control or indicator. The color of the assigned terminal in the connector pane darkens.

### 3.6.3 Deleting Terminal Connections

You can delete connections between terminals and the corresponding controls or indicators individually or all at once. Complete the following steps to delete a terminal connection.

**Step 1:** Right-click the terminal you want to disconnect on the connector pane and select *Disconnect This Terminal* from the shortcut menu.

**Step 2:** The terminal turns white to indicate that the connection no longer exists.

**Step 3:** *Disconnect This Terminal* is different from *Remove Terminal*, in that *Disconnect this Terminal* does not remove the terminal from the pattern.

**Step 4:** To delete all connections on the connector pane, right-click anywhere on the connector pane and select *Disconnect All Terminals* from the shortcut menu.

### 3.6.4 Setting Required, Recommended, and Optional Inputs and Outputs

You can designate which inputs and outputs are required, recommended, and optional to prevent users from forgetting to wire subVI terminals. Right-click a terminal in the connector pane and select *This Connection Is* from the shortcut menu. A checkmark indicates the terminal setting. Select *Required*, *Recommended* or *Optional*. For terminal inputs, *Required* means that the block diagram on which you placed the subVI will be broken if you do not wire the required inputs. *Required* is not available for terminal outputs. For terminal inputs and outputs, *Recommended* or *Optional* means that the block diagram on which you placed the subVI can execute if you do not wire the recommended or optional terminals. If you do not wire the terminals, the VI does not generate any warnings. Complete the following steps to set a terminal to required, recommended, or optional.

**Step 1:** Right-click a terminal in the connector pane and select *This Connection Is* from the shortcut menu.

**Step 2:** A checkmark indicates the terminal setting. Select *Required*, *Recommended* or *Optional*.

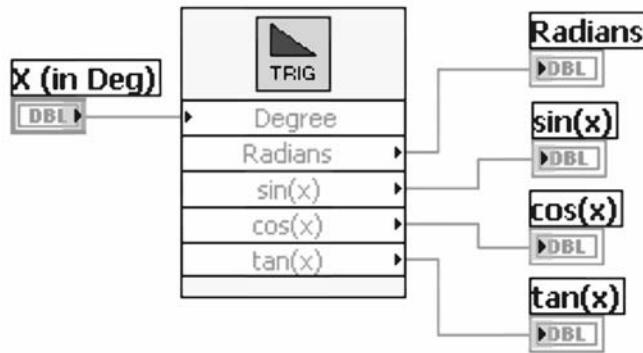
Inputs and outputs of VIs in vi.lib are already marked as *Required*, *Recommended* or *Optional*. LabVIEW sets inputs and outputs of VIs you create to *Recommended* by default. Set a terminal setting to *Required* only if the VI must have the input or output to run properly. In the *Context Help* window, the labels of required terminals appear bold, recommended terminals appear as plaintext, and optional terminals appear dimmed. The labels of optional terminals do not appear if you click the *Hide Optional Terminals* and *Full Path* button in the *Context Help* window.

## 3.7 DISPLAYING SUBVIs AND EXPRESS VIs AS ICONS OR EXPANDABLE NODES

You can display a subVI as icons or as an expandable node. Expandable nodes appear as icons surrounded by a colored field. Expanded nodes of SubVIs appear with a yellow field. Use icons if you want to conserve space on the block diagram. Use expandable nodes to make wiring easier and to aid in documenting block diagrams. By default, subVIs appear as icons on the block diagram. To display a subVI as an expandable node, right-click the subVI and remove the checkmark next to the *View As Icon* shortcut menu item. Figure 3.10 shows the expandable node of the trigonometry subVI. If you display a subVI as an expandable node, you cannot display the terminals for that node and you cannot enable database access for that node.

When you resize an expandable subVI, the input and output terminals of the subVI appear below the icon. Optional terminals appear with gray backgrounds. Recommended or required input or output terminals you do not display appear as input or output arrows in the colored field

that surrounds the subVI icon. If you wire to an optional terminal when the subVI is expanded, then resize the subVI so the optional terminal no longer appears in the expanded field, the optional terminal appears as an input or output arrow in the colored field. However, if you unwire the optional terminal, the input or output arrow does not appear.

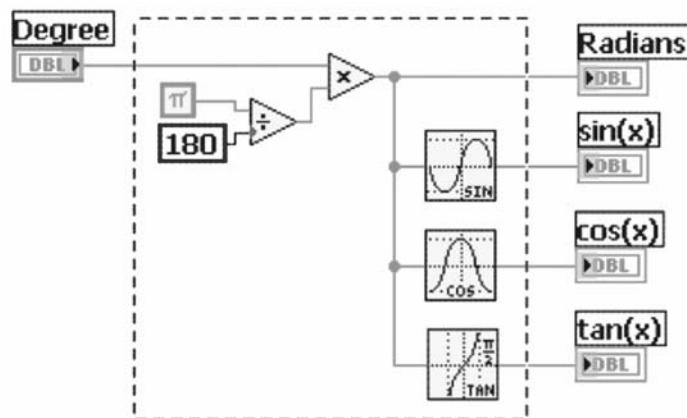


**Figure 3.10** SubVI displayed as expandable node.

By default, inputs appear above outputs when you expand the subVI. Right-click a terminal in the expandable field and select *Select Input/Output* from the shortcut menu to select an input or output to display. A line divides inputs from outputs in the shortcut menu. Labels for expandable subVIs appear in the colored field that surrounds the icon. To resize the expandable node so that it accommodates the name of each terminal on a single line in the expandable field, right-click the subVI and select *Size to Text* from the shortcut menu.

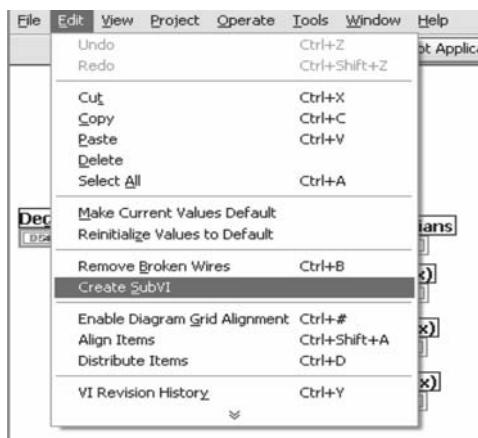
### 3.8 CREATING SUBVIs FROM SECTIONS OF A VI

You can convert a section of a VI into a subVI by using the positioning tool to select the section of the block diagram you want to reuse as shown in Figure 3.11. Then select *Edit»Create SubVI* as

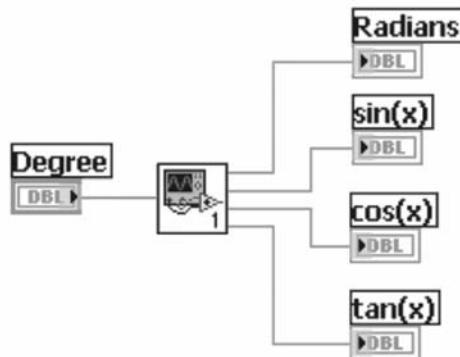


**Figure 3.11** Select a portion of the available VI to create a SubVI.

shown in Figure 3.12 from the menu to convert the selected portion into a subVI. SubVI created with default icon is shown in Figure 3.13. An icon for the new subVI replaces the selected section of the block diagram. LabVIEW creates controls and indicators for the new subVI, automatically configures the connector pane based on the number of control and indicator terminals you selected, and wires the subVI to the existing wires.



**Figure 3.12** Select *Edit»Create SubVI* to create subVI.



**Figure 3.13** SubVI created with default icon.

Creating a subVI from a selection is convenient but still requires careful planning to create a logical hierarchy of VIs. Consider which objects to include in the selection and avoid changing the functionality of the original or resulting VI. Creating a subVI from a selection is the same as removing the selected objects and replacing them with a subVI. In both situations, LabVIEW does not remove any of the front panel terminals included in the selection from the original VI. The front panel terminals remain on the front panel of the original VI and the terminals are wired to the new subVI. Also for each front panel object with a Property Node or local variable in the selection, LabVIEW adds a control reference to the original block diagram and wires the reference to the subVI. In the subVI, LabVIEW wires the reference to a Property Node. Select not more than

28 objects to create a subVI, because 28 is the maximum number of connections on a connector pane. You avoid creating cycles on the block diagram and including a structure that contains a front panel terminal in the selection.

### 3.9 OPENING AND EDITING SUBVIs

When you double-click a subVI, a front panel and a block diagram appear, rather than a dialog box in which you can configure options. The subVI controls and indicators receive data from and return data to the block diagram of the calling VI. Click the *Select a VI* icon or text on the *Functions* palette, navigate to and double-click a VI, and place the VI on a block diagram to call a created subVI.

Complete the following steps to open a subVI and edit it.

**Step 1:** Use the operating or positioning tool to double-click the subVI on the block diagram. LabVIEW displays the front panel of the subVI. You also can press the <Ctrl> key and use the operating or positioning tool to double-click the subVI on the block diagram to display the block diagram and front panel of the subVI. For Mac OS press the <Option> key and for Linux press the <Alt> key.

**Step 2:** Edit the subVI.

### 3.10 PLACING SUBVIs ON BLOCK DIAGRAMS

Complete the following steps to place a subVI on the block diagram.

**Step 1:** Display the block diagram of a new or existing VI by selecting *Window»Show Block Diagram*.

**Step 2:** If necessary, display the *Functions* palette by selecting *View»Functions Palette*.

**Step 3:** Click the *Select a VI* icon on the *Functions* palette.

**Step 4:** Navigate to and double-click the VI you want to use as a subVI, and place it on the block diagram.

**Step 5:** Wire the subVI terminals to other nodes on the block diagram.

### 3.11 SAVING SUBVIs

Select *File»Save* to save a VI. You can save VIs as individual files or you can group several VIs together and save them in an LabVIEW Library (LLB). LLB files end with the extension .llb. National Instruments recommends that we save VIs as individual files, organized in directories, especially if multiple developers are working on the same project.

### 3.12 CREATING A STAND-ALONE APPLICATION

Creating a stand-alone application and an installer simplifies deploying an application on multiple machines. In order to deploy the application, you first prepare the code, create an Application (Exe) Build Specification, and then create an Installer Build Specification.

A stand-alone application allows the user to run VIs without installing the LabVIEW development system. Installers are used to distribute the stand-alone application. Installers can include the LabVIEW Run-Time Engine, which is necessary for running stand-alone applications. However, you can also download the LabVIEW Run-Time Engine at ni.com.

To create a professional, stand-alone application with VIs, you must consider several programming issues. First, know what outside code your applications uses. For example, do you call any system or custom DLLs or shared libraries? Another issue is the path names you use in the VI. Assume you read data from a file during the application, and the path to the file is hard-coded on the block diagram. Once an application is built, the file is embedded in the executable, changing the path of the file. Being aware of these issues will help you build more robust applications in the future. Another issue that affects the application you have currently built is that the top-level VI does not quit LabVIEW or close the front panel when it is finished executing. To completely quit and close the top-level VI, you must call the Quit LabVIEW function on the block diagram of the top-level VI.

### 3.12.1 Building the Application and Installer

Use *Build Specifications* in LabVIEW to create stand-alone applications and installers.

*Stand-alone applications:* Use stand-alone applications to provide other users with executable versions of VIs. Applications are useful when you want users to run VIs without installing the LabVIEW development system. (Windows) Applications have a .exe extension. (Mac OS) Applications have a .app extension.

*Installers (Windows):* Use installers to distribute stand-alone applications, shared libraries, and source distributions that you create with the *Application Builder*. Installers that include the LabVIEW Run-Time Engine are useful if you want users to be able to run applications or use shared libraries without installing LabVIEW.

Use *Build Specifications* in the *Project Explorer* window to create build specifications for source distributions and other types of LabVIEW builds. A build specification contains all the settings for the build, such as files to include, directories to create and settings for directories of VIs.

### 3.12.2 Application (EXE) Build Specification

- Right-click *Build Specifications* in the *Project Explorer* window and select *New»Application (EXE)* from the shortcut menu.
- Modify the filename of the target and destination directory for the application in the *Application Information* category.
- Select the *Application Information* category.
- Change the *Target filename* to your filename.exe.

---

## SUMMARY

---

- Modularity increases the readability and reusability of your VIs.
- A VI within another VI is called modules or subVI.

- SubVIs correspond to a subroutine in text-based programming languages.
- The upper-right corner of the front panel and block diagram displays the icon for the VI.
- After you build a VI front panel and block diagram, build the icon and the connector pane to use the VI as a subVI.
- Right-click the icon in the upper-right corner of the front panel or block diagram and selecting *Edit Icon*, you can create custom icons to replace the default icon.
- Right-click the icon in the upper-right corner of the front panel and select *Show Connector*.
- The connector pane is a set of terminals that correspond to the controls and indicators of that VI. Define connections by assigning a front panel control or indicator to each of the connector pane terminals using the wiring tool.
- Load subVIs using the *Select a VI* option in the *All Functions* palette or dragging the icon onto a new diagram.
- Document a VI by selecting *File»VI Properties* and selecting *Documentation* from the *Category* pull-down menu. When you move the cursor over a VI icon, the *Context Help* window displays this description and indicates which terminals are required, recommended or optional.
- Add descriptions and tip strips to controls and indicators by right-clicking them and selecting *Description and Tip* from the shortcut menu. When you move the cursor over controls and indicators, the *Context Help* window displays this description.
- Convert a section of a VI into a subVI by using the *Positioning tool* to select the section of the block diagram you want to reuse and by selecting *Edit»Create SubVI*.
- LabVIEW features the *Application Builder*, which enables you to create stand-alone executables and installers. The *Application Builder* is available in the *Professional Development Systems* or as an add-on package.
- Creating a professional, stand-alone application with your VIs involves understanding the following:
  - ◆ The architecture of your application
  - ◆ The programming issues particular to the application
  - ◆ The application building process
  - ◆ The installer building process

---

## MISCELLANEOUS SOLVED PROBLEMS

---

**Problem 3.1** Create a VI to compute full adder logic using half adder logic as subVI.

**Solution** The front panel and the block diagrams for the half adder is shown in Figures P3.1(a) and P3.1(b). Create a subVI called “HALF” and use this in the main VI to create a full adder as given in Figures P3.1(c) and P3.1(d).

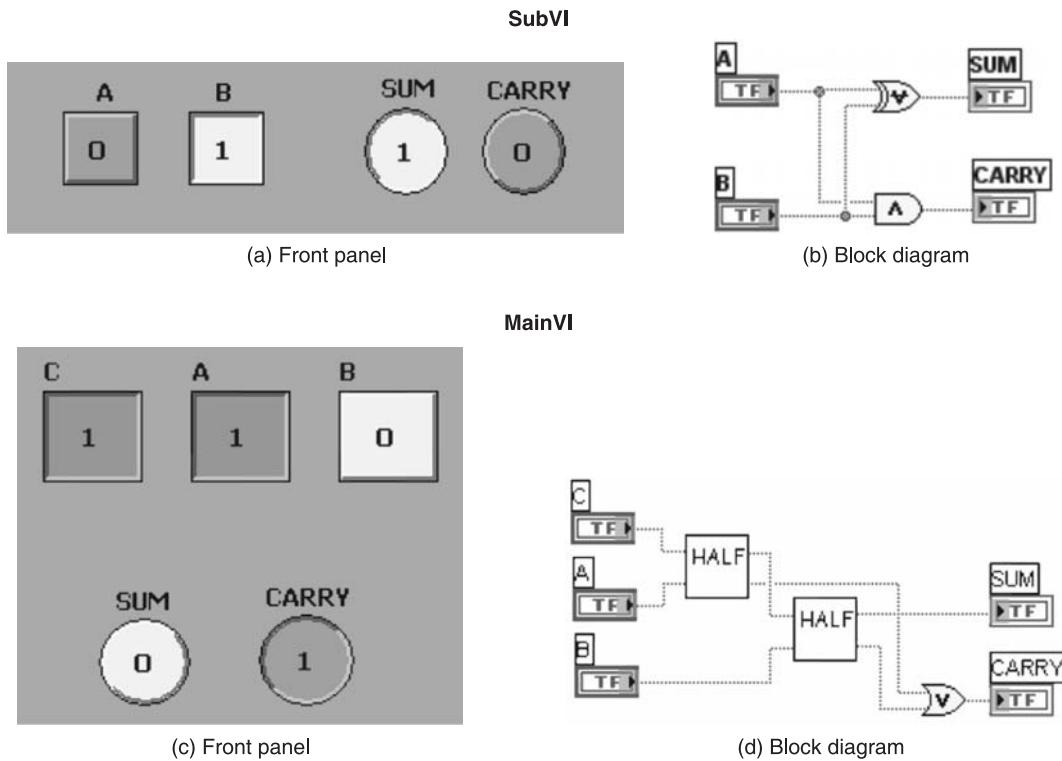


Figure P3.1

**Problem 3.2** Create a VI to find the decimal equivalent of a binary number using subVI.

**Solution** To find the decimal equivalent of a binary number, first create the front panel and the block diagram as given in Figures P3.2(a) and P3.2(b). Then create the subVI called “Binary to decimal” and use it in the main VI as shown in Figures P3.2(c) and P3.2(d).

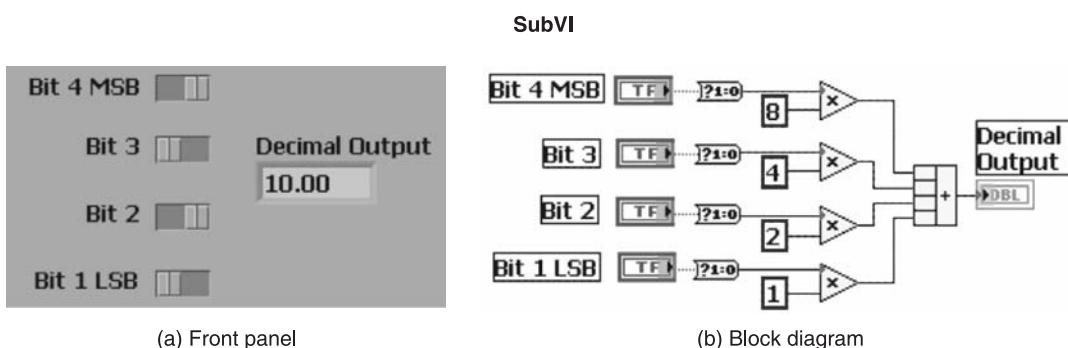


Figure P3.2 (Contd.)

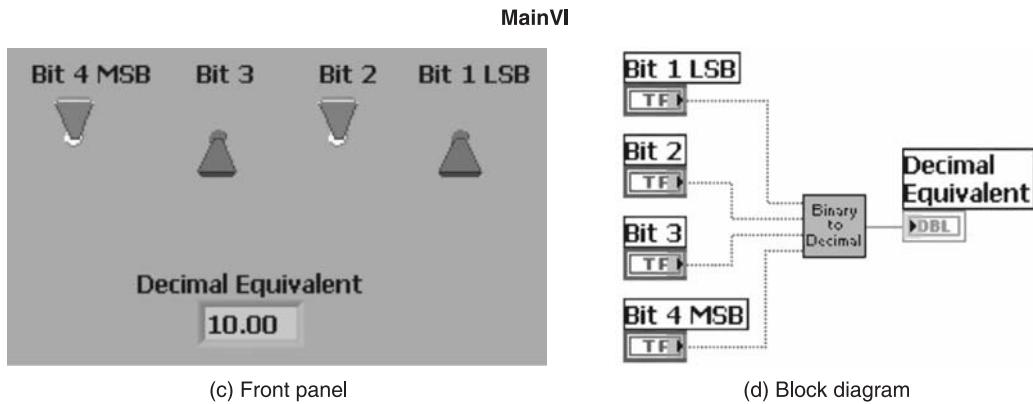


Figure P3.2

**Problem 3.3** Create a VI to find the Grey code equivalent of a BCD number using subVIs.

**Solution** Create the front panel and the block diagram as given in Figure P3.3(a) and P3.3(b). Then create the subVI called “BCD to GRAY” and use it in the main VI as shown in Figure P3.3(c) and P3.3(d).

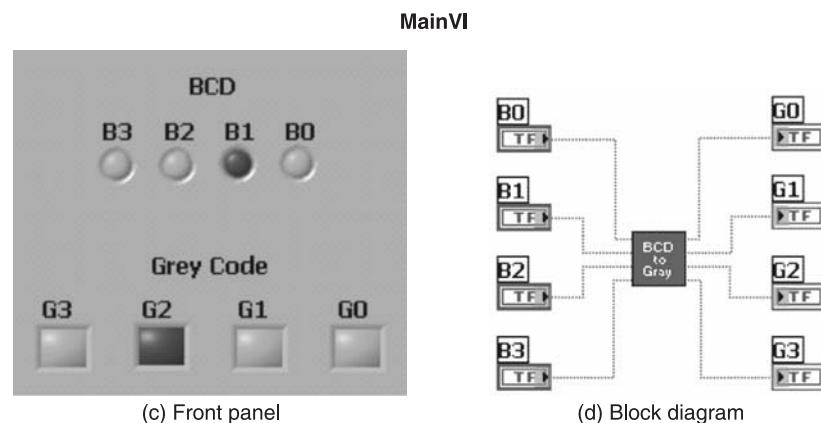
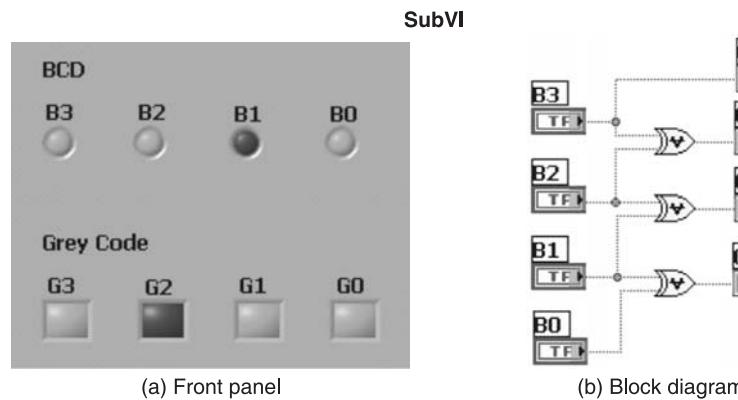
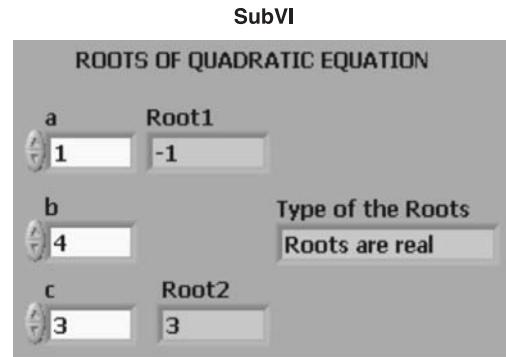


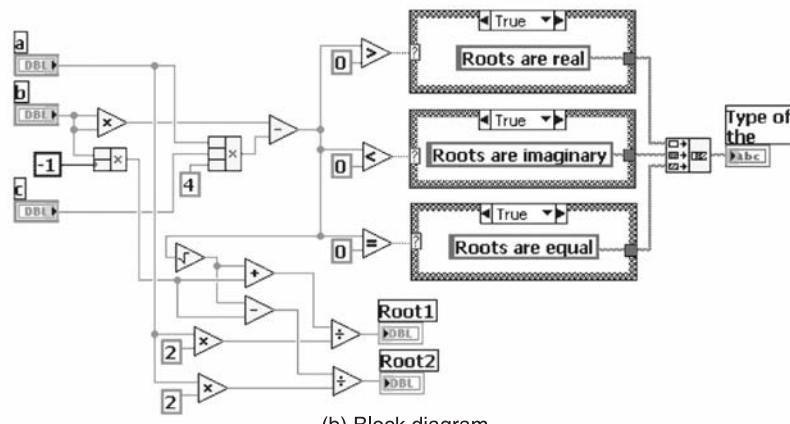
Figure P3.3

**Problem 3.4** Create a VI to find the roots of a quadratic equation using subVIs. Find both the values of the roots and the nature of the roots.

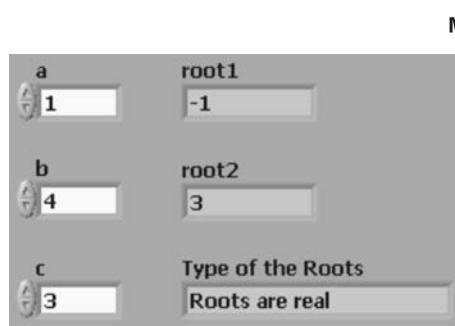
**Solution** Create the front panel and the block diagram as shown in Figures P3.4(a) and P3.4(b). Then create the subVI and use it in the main VI as shown in Figures P3.4(c) and P3.4(d).



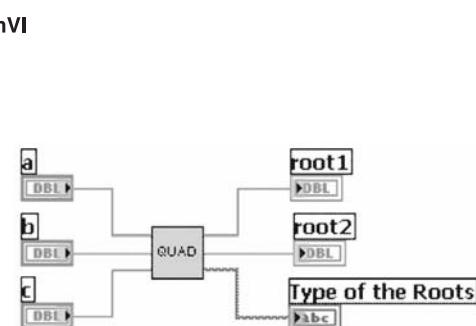
(a) Front panel



(b) Block diagram



(c) Front panel



(d) Block diagram

Figure P3.4

**Problem 3.5** Create a VI to find the average of two numbers and convert a section of a VI into a subVI.

**Solution** The block diagram to find the average of two numbers and convert a section of a VI into a subVI by selecting that portion is shown in Figures P3.5(a) and P3.5(b) respectively.

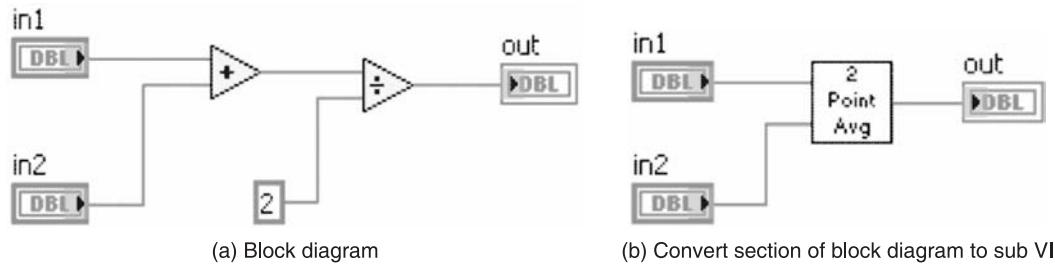


Figure P3.5

**Problem 3.6** Create the front panel and block diagram of the Main VI to show the trigonometric values (sine, cosine and tangent) of the given degree. This VI uses a subVI for finding the trigonometric values. The SubVI consists of functions for converting degree values to radians and functions to find the sine, cosine and tangent values separately. In LabVIEW the sine, cosine and tangent functions take input in radians.

**Solution** Create the front panel and the block diagram as shown in Figures P3.6(a) and P3.6(b).

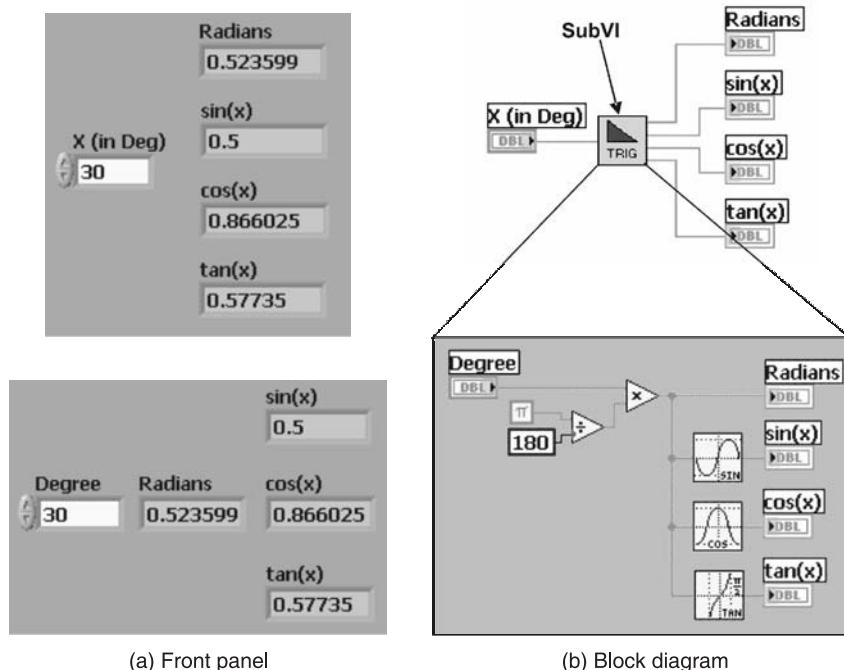


Figure P3.6

### **REVIEW QUESTIONS**

---

1. What is modular programming?
2. Define subVI in LabVIEW?
3. Explain the need of icon and connector pane.
4. What is a default icon and a custom icon?
5. Consider an example and explain the steps involved in creating, saving and retrieving subVI?
6. Explain how to create setting required, recommended, and optional inputs and outputs in creating terminals of a SubVI.
7. What is the difference between viewing subVIs as icons and expandable nodes?
8. How to create a subVI from the portion of an available VI?
9. How can a stand-alone application be created?
10. How can subVI be displayed as expandable nodes?

### **EXERCISES**

---

1. Create a VI that takes a number representing Celsius and converts it to a number representing Fahrenheit. Build the subVI.
2. Build a subVI of the VI that accepts a user name and password and checks for a match in a table of information of two employees. If the name and password match, the confirmed name and a verification Boolean object are returned.
3. Create a subVI to compute the average of five students marks.
4. Given the length and breadth, build a subVI to compute the area of the polygon.
5. Build a subVI to calculate the gross salary, given the basic salary as input. The dearness allowance is 35% of basic salary and house rent allowance is 25% of basic salary.
6. Create a subVI given two inputs  $x$  and  $y$ , and compute the complex number, its complex conjugate and polar components.
7. Build a subVI to convert radians to degrees.



## REPETITION AND LOOPS

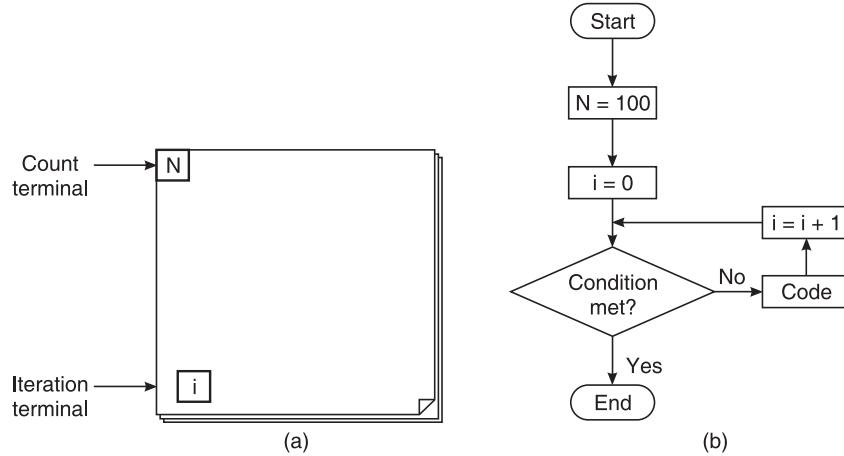
---

### 4.1 INTRODUCTION

Loops and case statements of text-based programming languages are represented as structures in graphical programming. Repetition and loop are used to perform an action frequently with variations in the details each time. LabVIEW consists of FOR Loop and WHILE Loop. These loops are used to control repetitive operations. Structures on the block diagram are used to repeat blocks of code and to execute code conditionally or in a specific order. LabVIEW includes structures like the While Loop, For Loop, Case structure, Stacked Sequence structure, Flat Sequence structure, Event structure, and Formula Node. This chapter introduces the loops in LabVIEW, iterative data transfer in loops and timing a loop along with functions commonly used with these structures, including the shift register and the Feedback Node.

### 4.2 FOR LOOPS

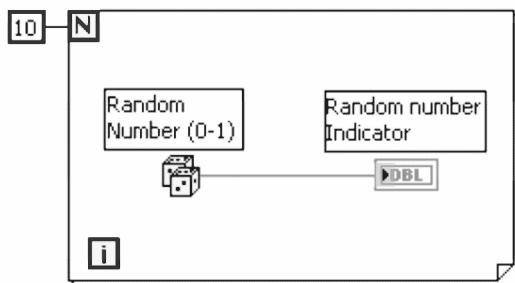
A For Loop executes a subdiagram, a set number of times. Figure 4.1(a) shows a For Loop in LabVIEW and Figure 4.1(b) shows the flow chart equivalent of the For Loop functionality. The For Loop is located on the *Functions>>Programming>>Structures Palette*. Select the For Loop from the palette and use the cursor to drag a selection rectangle to create a new For Loop or around the section of the block diagram you want to repeat. You also can place a While Loop on the block diagram, right-click the border of the While Loop, and select *Replace with For Loop* from the shortcut menu to change a While Loop to a For Loop.



**Figure 4.1** (a) For Loop in LabVIEW and (b) Flow chart equivalent to the For Loop.

**N** The value in the count terminal ‘N’ (an input terminal) indicates how many times to repeat the subdiagram. Set the count explicitly by wiring a value from outside the loop to the left or top side of the count terminal, or set the count implicitly with auto-indexing. Auto-indexing is explained in chapter 5. A VI will not run if it contains a For Loop that does not have a numeric value wired to the count terminal.

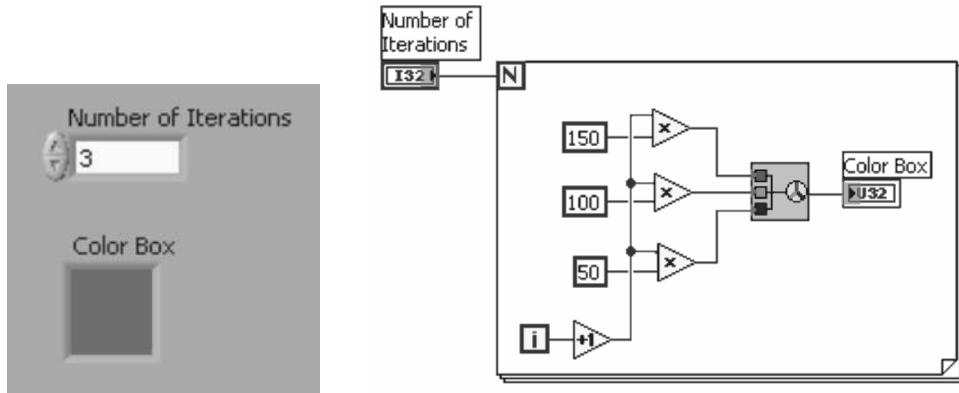
**i** The iteration terminal ‘i’ (an output terminal) contains the number of completed iterations. The iteration count always starts at zero. During the first iteration, the iteration terminal returns 0. Figure 4.2 shows a simple For Loop which generates 10 random numbers and displays in the *Random Number Indicator*.



**Figure 4.2** Generating random numbers using For Loop.

Both the count and iteration terminals are 32-bit signed integers. If you wire a floating-point number to the count terminal, LabVIEW rounds it and coerces it to within range. If you wire 0 or a negative number to the count terminal, the loop does not execute and the outputs contain the default data for that data type. A For Loop can only execute an integer a number of times.

**Example 4.1:** Create a VI using For Loop which changes the color of a color box automatically for the given number of iterations as shown in Figure 4.3.



**Figure 4.3** Example 4.1: Front panel and block diagram.

**Solution:** The front panel of this VI consists of a Numeric Control for providing number of iterations and a Framed Color Box for displaying various colors as shown in the figure. The framed *Color Box* indicator is located in *Controls>>Modern>>Numeric* palette.



In the block diagram the function *RGB to Color* is used to produce various colors based on the combination of values given to the red, green and blue terminals. The *RGB to Color* function is located in *Functions>>Programming>>Numeric>>Conversion* palette.

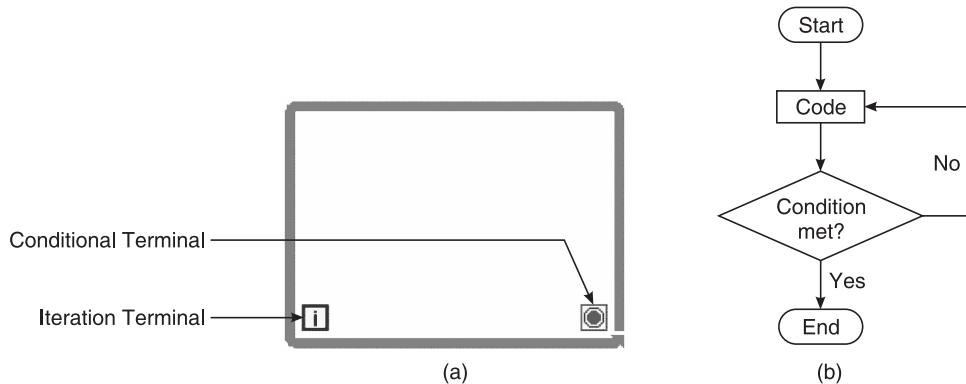
As given in the solution, the For Loop will execute for three iterations. The value of the iteration terminal added with 1 is given as input for one of the terminals of multiplication functions for each iteration. The iteration value is multiplied with 50, 100 and 150 for producing red, green and blue values respectively. These values are given to the *RGB to Color* function. This *RGB to Color* function produces three different colors based on the red, green and blue values. The produced colors are displayed in the framed *Color Box* indicator. The For Loop will execute only for three iterations since the number of iterations is set to 3. The number of iterations can be increased to produce more numbers of colors.

### 4.3 WHILE LOOPS

A While Loop executes a subdiagram until a condition is met. The While Loop is similar to a Do Loop or a Repeat-Until Loop in text-based programming languages. Figure 4.4(a) shows a While Loop in LabVIEW and 4.4(b) is the flow chart equivalent of the While Loop. The While Loop always executes at least once. The For Loop differs from the While Loop in that the For Loop executes a set number of times. A While Loop stops executing the subdiagram, only if the expected value at the conditional terminal exists.

In LabVIEW, the WHILE Loop is located on the *Functions>>Programming>>Structures* palette. You also can place a For Loop on the block diagram, right-click the border of the For Loop, and select *Replace with While Loop* from the shortcut menu to change a For Loop to a While Loop. The While Loop contains two terminals, namely *Conditional Terminal* and *Iteration Terminal*.

The Conditional Terminal is used to control the execution of the loop, whereas the Iteration Terminal is used to know the number of completed iterations.



**Figure 4.4** (a) While Loop in LabVIEW and (b) Flow chart equivalent to the While Loop.

Conventional programming languages support two types of WHILE constructs as illustrated in Figure 4.5. These are called pre- and post-test modes. In the pre-test mode the condition is tested prior to the execution of every iteration and if the result is false, then the execution of the loop is aborted. In the post-test mode the test is carried out only at the end of the loop. Functionally, the major difference is that under the post-test mode even if the condition is false at the first execution or first iteration, the loop will be executed at least once, since the test is only performed at the end of the loop. LabVIEW supports only the post-test form of the While construct.

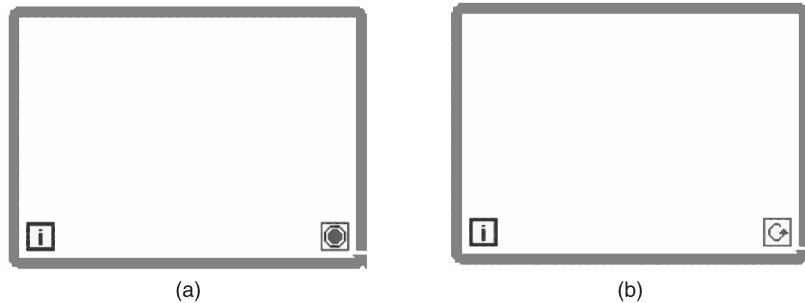


**Figure 4.5** (a) Pre-test mode and (b) Post-test modes of a While loop.

The While Loop executes the subdiagram until the conditional terminal, and receives a specific Boolean value. The default behavior and appearance of the conditional terminal is *Stop if True* as shown in Figure 4.6(a). When a conditional terminal is *Stop if True*, the While Loop executes its subdiagram until the conditional terminal receives a TRUE value. You can change the behavior and appearance of the conditional terminal by right-clicking the terminal or the border of the While Loop and selecting *Continue if True* from the shortcut menu as shown in Figure 4.6b). When a conditional terminal is *Continue if True*, the While Loop executes its subdiagram until the conditional terminal receives a FALSE value. You also can use the Operating Tool to click the conditional terminal to change the condition. The VI shows error if the conditional terminal is unwired.

You also can perform basic error handling using the conditional terminal of a While Loop. When you wire an error cluster to the conditional terminal, only the TRUE or FALSE value of the status parameter of the error cluster passes to the terminal. Also, the *Stop if True* and *Continue if*

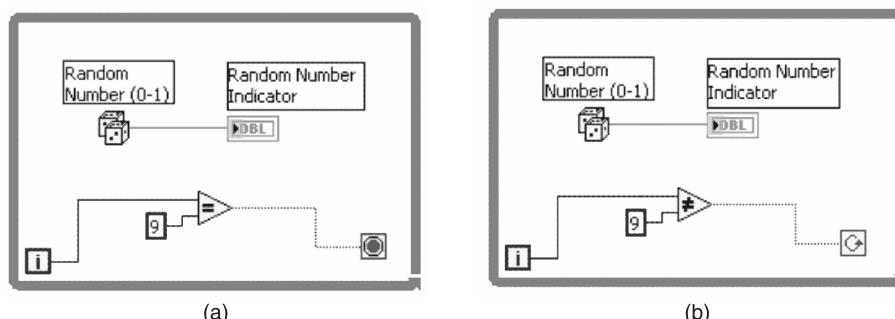
*True* shortcut menu items change to *Stop if Error* and *Continue While Error*. Error clusters are explained in Chapter 6 in detail.



**Figure 4.6** (a) While Loop with conditional terminal as *Stop if True* and  
(b) While Loop with conditional terminal as *Continue if True*.

**i** The iteration terminal ‘i’ (an output terminal), contains the number of completed iterations. The iteration count always starts at zero. During the first iteration, the iteration terminal returns 0. Figure 4.7 shows a simple While Loop which generates 10 random numbers and displays in the Random Number Indicator.

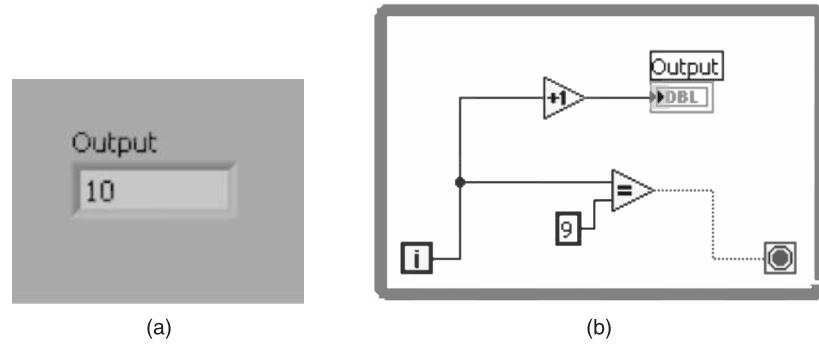
The block diagrams shown in Figures 4.7(a) and 4.7(b) consist of a random number generator function. The While Loop will execute for 10 iterations (where ‘i’ value ranges from 0 to 9). A random number is generated and displayed for each iteration in the Numeric Indicator in the Front Panel.



**Figure 4.7** (a) Generating random numbers using While Loop with conditional terminal as *Stop if True*, and (b) Generating random numbers using While Loop with conditional terminal as *Continue if True*.

In the block diagram shown in Figure 4.7(a), the conditional terminal used is *Stop if True*. When the iteration value ‘i’ reaches 9, the conditional terminal receives a TRUE state and stops executing the loop. In the block diagram shown in Figure 4.7(b), the conditional terminal used is *Continue if True*. For the iteration value ‘i’ ranges from 0 to 8, the conditional terminal receives a TRUE state and keeps the loop executing. When the iteration value becomes 9, the conditional terminal receives a FALSE state and stops executing the loop.

**Example 4.2:** Create a VI to display the numbers 1 to 10 in a Numeric Indicator using a While Loop. Use the conditional terminal *Stop if True* as in Figure 4.8.

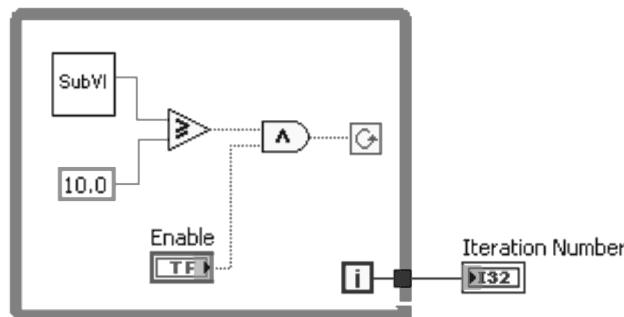


**Figure 4.8** Example 4.2: Front panel and block diagram to display numeric indicator.

**Solution:** The front panel consists of a Numeric Indicator to display the numbers. In the block diagram the While Loop executes for 10 iterations where the iteration value ‘i’ ranges from 0 to 9. When the iteration value reaches 9, the conditional terminal receives a TRUE state and stops the execution of the While Loop. The iteration value is added with 1 to produce output from 1 to 10 and it is displayed in the Numeric Indicator.

#### 4.4 STRUCTURE TUNNELS

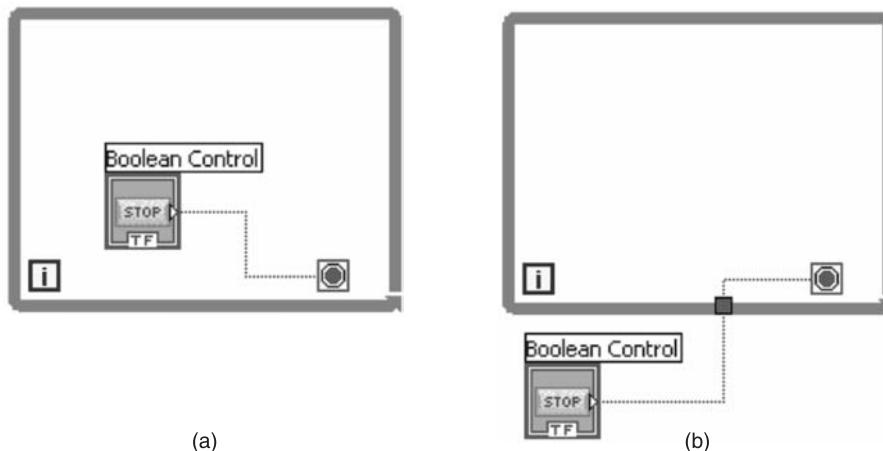
Data can be passed out of or into a loop through a tunnel. Tunnels feed data into and out of structures. The tunnel appears as a solid block on the border of the loop. The block is the color of the data type wired to the tunnel. Data passes out of a loop after the loop terminates. When a tunnel passes data into a loop, the loop executes only after data arrives at the tunnel. In Figure 4.9, the iteration terminal is connected to a tunnel. The value in the tunnel does not pass to the *Iteration Number* indicator until the While Loop has finished execution. Only the last value of the iteration terminal displays in the *Iteration Number* indicator.



**Figure 4.9** Passing data outside the loop through a tunnel.

## 4.5 TERMINALS INSIDE OR OUTSIDE LOOPS

Inputs pass data into a loop at the start of loop execution. Outputs pass data out of a loop only after the loop completes all iterations. If you want the loop to check the value of a terminal for each iteration, place the terminal inside the loop. When you place the terminal of a front panel Boolean control inside a While Loop and wire the terminal to the conditional terminal of the loop, the loop checks the value of the terminal for every iteration to determine if it must iterate. You can stop the While Loop as shown in Figure 4.10(a) by changing the value of the front panel control to FALSE.



**Figure 4.10** (a) While Loop with Boolean control placed inside the loop and (b) While Loop with Boolean control placed outside the loop.

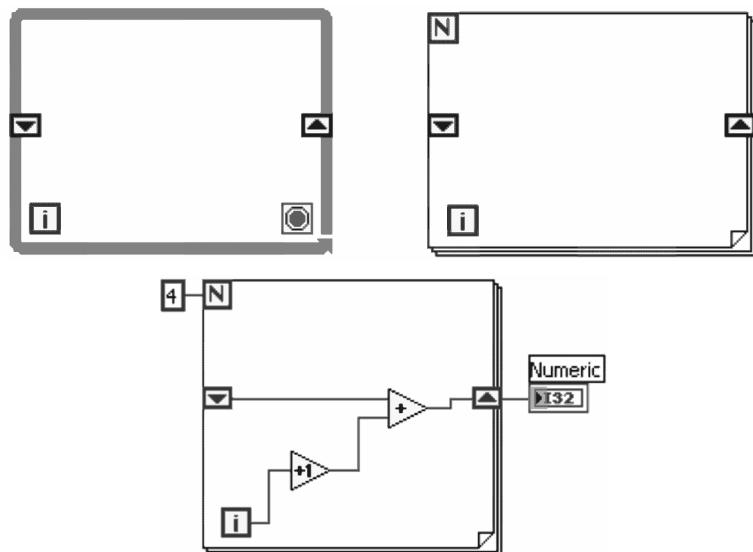
If you place the terminal of the Boolean control outside the While Loop as shown in Figure 4.8(b), and the control is set to FALSE if the conditional terminal is *Stop if True* when the loop starts, you cause an infinite loop. You also cause an infinite loop if the control outside the loop is set to TRUE and the conditional terminal is *Continue if True*. Changing the value of the control does not stop the infinite loop because the value is only read once, before the loop starts. To stop an infinite loop, you must abort the VI by clicking the *Abort Execution* button on the toolbar.

## 4.6 SHIFT REGISTERS

When programming with loops, you often need to access data from previous iterations of the loop. For example, you may have a VI that reads the temperature and displays it on a graph. If you want to display a running average of the temperature as well, you need to use data generated in previous iterations. Two ways of accessing this data include the shift register and the feedback node.

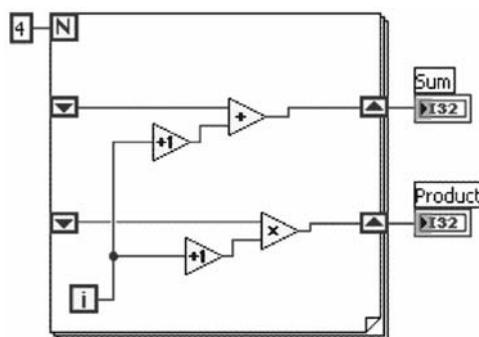
While Loops produce default data when the shift register is not initialized. For Loops produce default data if you wire 0 to the count terminal of the For Loop or if you wire an empty array to the For Loop as an input with auto-indexing enabled. The loop does not execute, and any output tunnel with auto-indexing disabled contains the default value for the tunnel data type. Use shift registers to transfer values through a loop regardless of whether the loop executes.

Shift registers are used with For Loops and While Loops to transfer values from one loop iteration to the next. Shift registers are similar to static variables in text-based programming languages. A shift register appears as a pair of terminals, shown in Figure 4.11, directly opposite each other on the vertical sides of the loop border. The terminal on the right side of the register contains an up arrow and stores data on the completion of an iteration. LabVIEW transfers the data connected to the right side of the register to the next iteration. After the loop executes, the terminal on the right side of the loop returns the last value stored in the shift register. Create a shift register by right-clicking the left or right border of a loop and selecting the *Add shift register* from the shortcut menu.



**Figure 4.11** Shift register operation.

A shift register transfers any data type and automatically changes to the data type of the first object wired to the shift register. The data you wire to the terminals of each shift register must be the same type. You can add more than one shift register to a loop. If you have multiple operations that use previous iteration values within our loop, you can use multiple shift registers to store the data values from those different processes in the structure as shown in Figure 4.12.



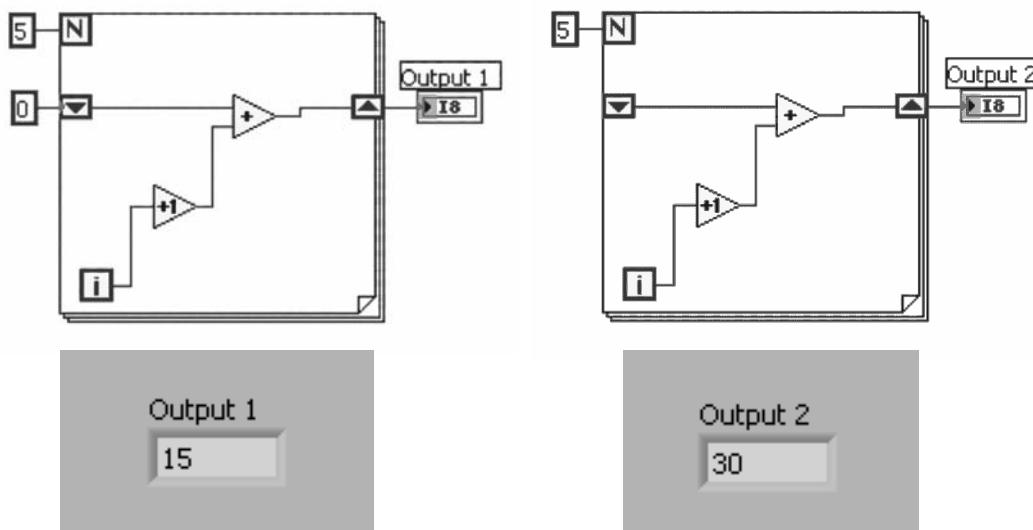
**Figure 4.12** Multiple shift registers.

#### 4.6.1 Initializing Shift Registers

Initializing the shift register resets the value the shift register passes to the first iteration of the loop when the VI runs. Initialize a shift register by wiring a control or constant to the shift register terminal on the left side of the loop. If you do not initialize the shift register, the loop uses the value written to the shift register when the loop last executed or the default value for the data type if the loop has never executed.

Use a loop with an uninitialized shift register to run a VI repeatedly so that each time the VI runs, the initial output of the shift register is the last value from the previous execution. Use an uninitialized shift register to preserve state information between subsequent executions of a VI. After the loop executes, the last value stored in the shift register remains at the right terminal. If you wire the right terminal outside the loop, the wire transfers the last value stored in the shift register. You can add more than one shift register to a loop.

If you have multiple operations that use previous iteration values within a loop, use multiple shift registers to store the data values from those different processes in the structure as shown in Figure 4.13. The For Loop executes five times. The shift register value is added with the subsequent numbers starting from 1 to 5. After five iterations of the For Loop, the shift register passes the final value 15 to the indicator and the VI stops. Each time you run the VI, the shift register begins with a value of 0. The shift register is used without initialization. The first time you run the VI, the shift register begins with a value of 0, which is the default value for a 32-bit integer. After five iterations of the For Loop, the shift register passes the final value 15 to the indicator and ends. The next time you run the VI, the shift register begins with a value of 15, which was the last value from the previous execution. After five iterations of the For Loop, the shift register passes the final value 30 to the indicator. If you run the VI again, the shift register begins with a value of 30, and so on. Uninitialized shift registers retain the value of the previous iteration until you close the VI.



**Figure 4.13** Initializing shift registers.

#### 4.6.2 Stacked Shift Registers

Stacked shift registers let you access data from previous loop iterations. Stacked shift registers remember values from multiple previous iterations and carry those values to the next iterations. This technique is useful for averaging data points. To create a stacked shift register, right-click the left terminal and select *Add Element* from the shortcut menu.

Stacked shift registers, as shown in Figure 4.14, can only occur on the left side of the loop, because the right terminal only transfers the data generated from the current iteration to the next iteration. If you add one more element to the left terminal, the value from the last two iterations carry over to the next iteration, with the most recent iteration value stored in the top shift register. The second terminal stores the data passed to it from the previous iteration. You can add more than two terminals in the shift register. If you add two more elements to the left terminal, the second terminal stores the data passed to it from the previous iteration and the bottom terminal stores the data passed to it from two iterations ago.

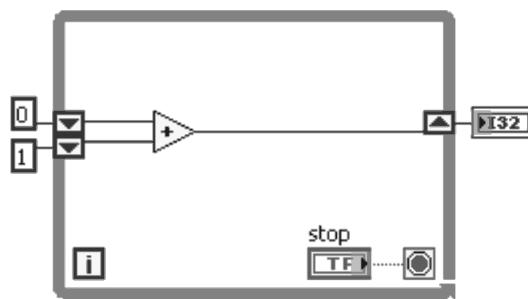


Figure 4.14 Stacked shift registers.

#### 4.6.3 Replacing Tunnels with Shift Registers

Tunnels can be replaced with shift registers wherever necessary. To replace a tunnel into a shift register, right-click the tunnel and select *Replace with Shift Register*. If no tunnel exists on the loop border opposite of the tunnel you right-clicked, LabVIEW automatically creates a pair of shift register terminals. If one or more than one tunnel exists on the loop border opposite of the tunnel you right-clicked, the mouse pointer will turn to the symbol of a shift register. You can choose the particular tunnel which is to be converted as the shift register by clicking the mouse over that tunnel as shown in Figure 4.15.

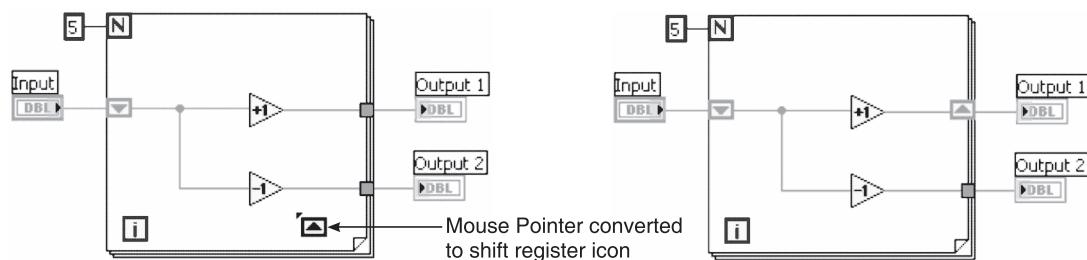
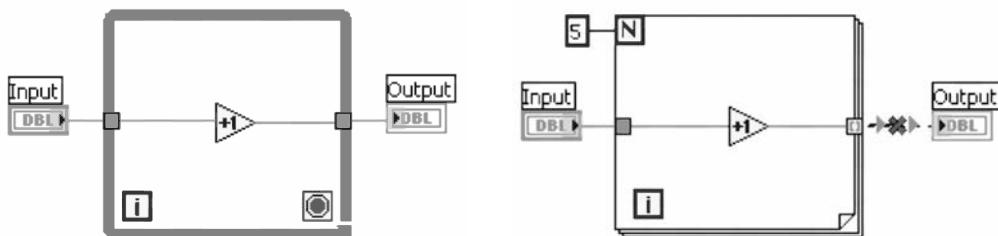


Figure 4.15 Replace a tunnel into a shift register.

#### 4.6.4 Replacing Shift Registers with Tunnels

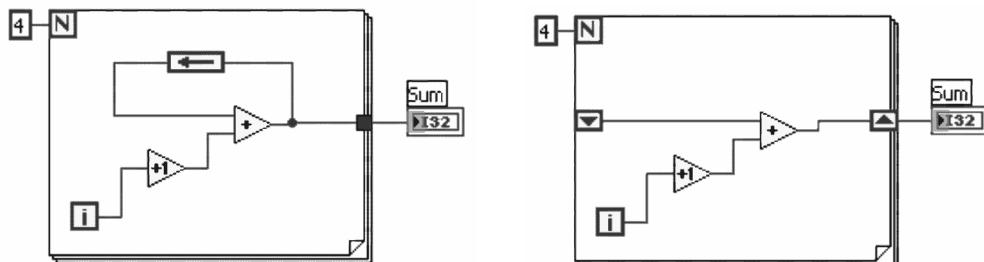
Replace shift registers with tunnels when you no longer need to transfer values from one loop iteration to the next. To replace a shift register with a tunnel, right-click the shift register and select *Replace with Tunnels*. If you replace an output shift register terminal with a tunnel on a For Loop, the wire to any node outside the loop breaks because the For Loop enables auto-indexing by default as shown in Figure 4.16. Right click the auto-indexed tunnel and select *Disable Indexing* on the tunnel to correct the broken wire. This problem does not occur in While Loops because auto-indexing is disabled by default in While Loops. The concept of auto-indexing is explained in Chapter 5 in detail.



**Figure 4.16** Replace shift registers with tunnels.

#### 4.7 FEEDBACK NODES

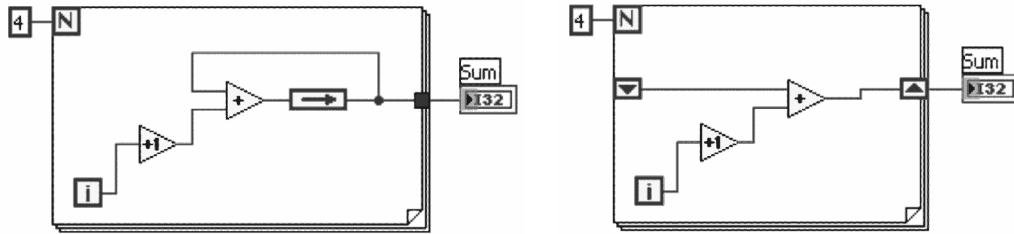
When the output of a node is connected directly to the input, the feedback node is generated automatically. The feedback node shown in Figure 4.17 appears automatically in a For Loop or While Loop if we wire the output of a node or group of nodes to the input of that node or group of nodes. Like a shift register, the feedback node stores data when the loop completes an iteration, sends that value to the next iteration of the loop, and transfers any data type. Use the feedback node to avoid unnecessarily long wires in loops. The feedback node arrow indicates the direction in which the data flows along the wire. The arrow automatically changes direction if the direction of data flow changes.



**Figure 4.17** Feedback node.

You also can select the *Feedback Node* on the *Structures* palette and place it inside a For Loop or While Loop. If you place the feedback node on the wire before you branch the wire that

connects the data to the tunnel as in Figure 4.18, the feedback node passes each value to the tunnel. If you place the feedback node on the wire after you branch the wire that connects data to the tunnel, the feedback node passes each value back to the input of the VI or function and then passes the last value to the tunnel.

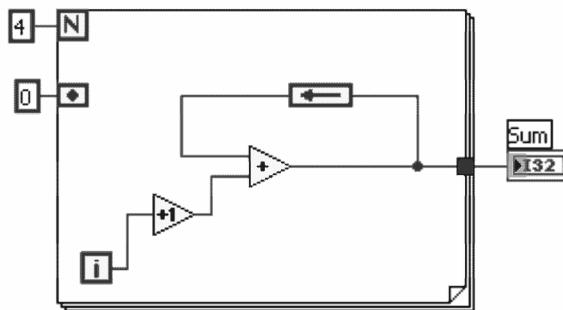


**Figure 4.18** Use of feedback node.

#### 4.7.1 Initializing a Feedback Node

Initializing a feedback node as shown in Figure 4.19 resets the initial value that the feedback node passes for the first time the loop executes when the VI runs. If you do not initialize the feedback node, the feedback node passes the last value written to the node or the default value for the data type if the loop has never executed. If you leave the input of the initializer terminal unwired, each time the VI runs, the initial input of the feedback node is the last value from the previous execution.

To initialize a feedback node, right-click the *Feedback Node* and select *Initializer Terminal* from the shortcut menu and add wire a value from outside the loop to the initializer terminal. When you select the *Feedback Node* on the *Functions* palette or if you convert an initialized shift register to a feedback node, the loop appears with an initializer terminal.



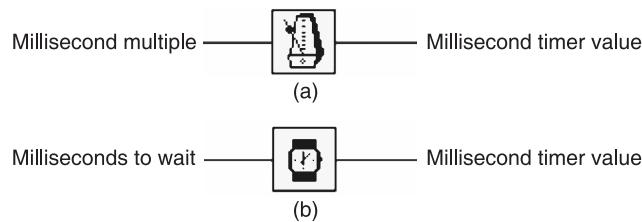
**Figure 4.19** Initializing a feedback node.

To replace a feedback node with shift registers, right-click the *Feedback Node* and select *Replace with Shift Register* from the shortcut menu.

To replace shift registers with a feedback node, right-click the *Shift Registers* and select *Replace with Feedback Node* from the shortcut menu.

## 4.8 CONTROL TIMING

When a loop finishes executing an iteration, it immediately begins executing the next iteration unless it reaches a stop condition. Most applications need precise control of the frequency or timing of the iteration to be maintained between successive operations of the loop. You might want to control the speed at which a process executes, such as the speed at which data values are plotted to a chart. You can use a wait function in the loop to wait an amount of time in milliseconds before the loop re-executes. LabVIEW consists of two wait functions. A wait function is placed inside a loop to allow a VI to sleep for a set amount of time. This allows your processor to address other tasks during the wait time. Wait functions use the operating system millisecond clock. They are *Wait Until Next ms Multiple* as shown in Figures 4.20(a) and *Wait (ms)* functions as shown in Figure 4.20(b).



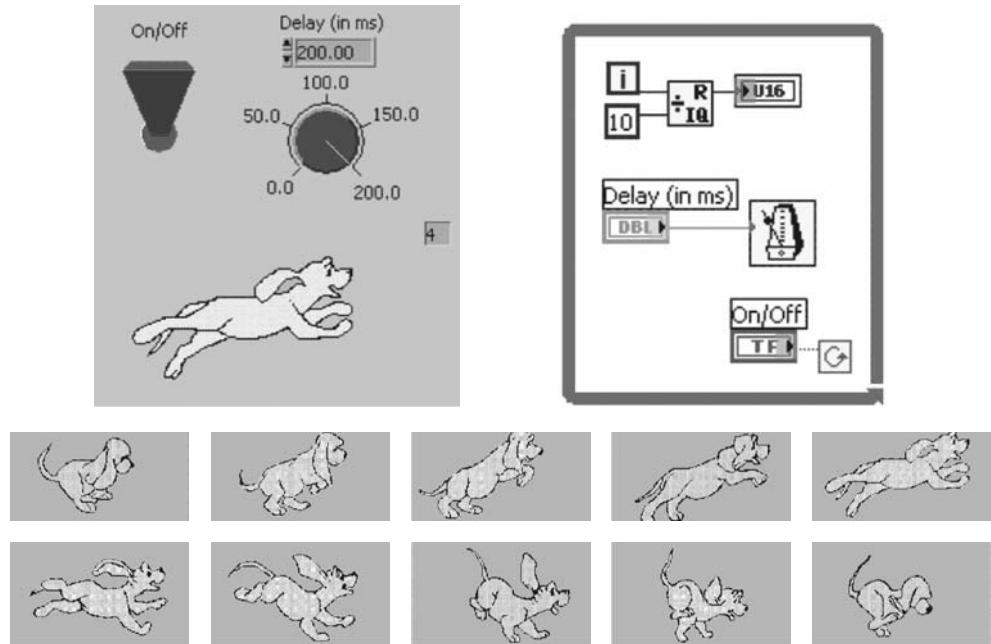
**Figure 4.20** (a) Wait Until Next ms Multiple function and (b) Wait (ms) function.

The *Wait Until Next ms Multiple* function monitors a millisecond counter and waits until the millisecond counter reaches a multiple of the time you specify. This function can be used for synchronization activities. You can place this function within a loop to control the loop execution rate. For this function to be effective, your code execution time must be less than the time specified for this function. The execution rate for the first iteration of the loop is indeterminate.

The *Wait (ms)* function adds the wait time to the code execution time. This can cause a problem if code execution time is variable. The *Wait (ms)* function waits until the millisecond counter counts to an amount equal to the input you specify. This function guarantees that the loop execution rate is at least the amount of the input you specify. The *Time Delay Express* VI, located on the *Functions>>Execution Control* palette, behaves similar to the *Wait (ms)* function with the addition of built-in error clusters.

**Example 4.3:** Animate a dog running using *Picture Ring Control* and *Wait Until Next ms Multiple* function as shown in Figure 4.21.

**Solution:** The front panel consists of a Boolean control to ON or OFF the VI execution, a numeric control to adjust the time delay and a Picture Ring Indicator to display the pictures. To animate an action, you need a sequence of pictures with all the actions. In this example 10 different pictures consists of 10 different actions of a dog at running state is taken for animation. Following are the pictures taken for animation. If these pictures are animated, it shows as if a dog is running.



**Figure 4.21** Example 4.3: front panel and block diagram.

In the front panel, the *Picture Ring Indicator* is used to display sequence of pictures. The *Picture Ring* is a control by default but you have to convert it to indicator for this VI. The *Picture Ring Control* is available in the location *Controls>>Modern>>Ring & Enum* palette. Following are the steps to insert picture in the *Picture Ring Control*.

**Step 1:** Place the *Picture Ring Control* in the front panel and change it to the *Picture Ring Indicator*.

**Step 2:** In the front panel menu go to *Edit* and choose *Import Picture From File ...*

**Step 3:** A file dialog appears in which you can choose the picture. Now the picture is available in *Windows Clipboard*.

**Step 4:** Right-click the *Picture Ring Indicator* and choose *Import Picture From Clipboard*. The picture available in the clipboard will be inserted in the *Picture Ring Indicator*.

**Step 5:** To insert the next picture follow steps 2 and 3. Right-click the *Picture Ring Indicator* and choose *Add Item After*. Now the new picture available in the clipboard will be inserted in the *Picture Ring Indicator* next to the previous picture.

**Step 6:** By followings steps 2, 3 and 5 you can insert all the 10 pictures. You can choose *Add Item After* or *Add Item Before* to insert pictures in appropriate order. After inserting all the pictures the order of the pictures, will be in the range of 0 to 9.

In the block diagram the value of the iteration terminal is divided by 10 because the number of pictures added is 10. The remainder of the quotient and remainder function varies from 0 to 9.

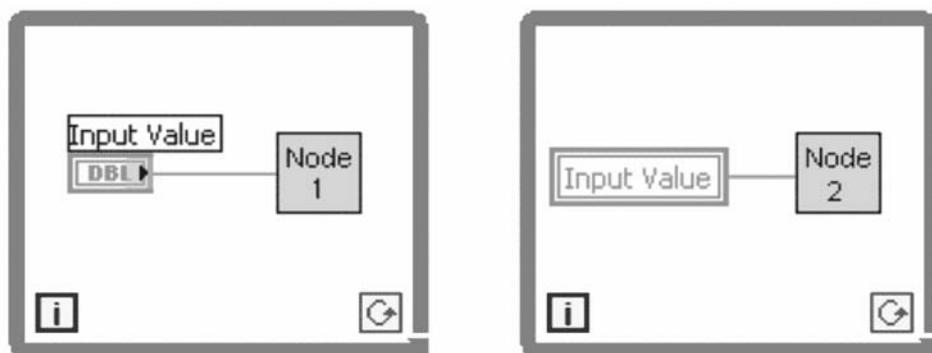
The remainder value is connected to the *Picture Ring Indicator*. If the remainder is 0, the first picture is selected and displayed; if the remainder is 1, the second picture is selected and displayed; and if the remainder is 9, the last picture, i.e. the 10th picture will be selected and displayed. The numeric control used to provide time delay is set in the range of 0 to 200 milliseconds. By varying the value of the numeric control, we can change the speed at which the pictures are displayed which intern changes the speed at which the dog runs. The Boolean control is used to run or stop the VI. For every iteration, a picture is displayed and that produces the animation of the dog running.

## 4.9 COMMUNICATING AMONG MULTIPLE LOOPS

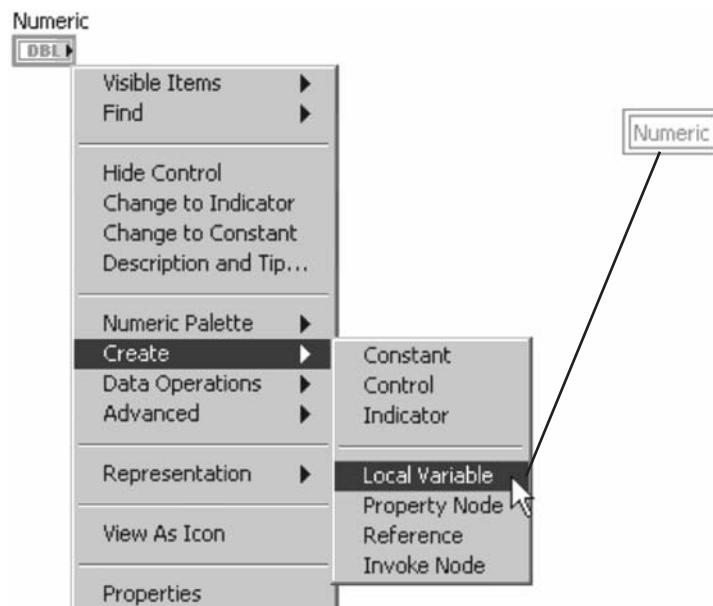
In LabVIEW, the flow of data determines the execution order of block diagram elements. *Variables* are block diagram elements that allow you to access or store data in another location. The actual location of the data varies depending upon the type of the variable. Local variables store data in front panel controls and indicators. Global variables and single process-shared variables store data in special repositories that you can access from multiple VIs. Functional global variables store data in While Loop shift registers. Regardless of where the variable stores data, all variables allow you to circumvent normal dataflow by passing data from one place to another without connecting the two places with a wire. For this reason, variables are useful in parallel architectures, but also have certain drawbacks, such as race conditions. You can create block diagrams that have simultaneous operations. But if you use wires to pass data between parallel block diagrams, they no longer operate in parallel. Parallel block diagrams can be two parallel loops on the same block diagram without any data flow dependency or two separate VIs that are called at the same time.

## 4.10 LOCAL VARIABLES

Local variables transfer data within a single VI and allow data to be passed between parallel loops as shown in Figure 4.22. They also break the dataflow programming paradigm. Two ways to create a local variable are right-click on an object's terminal and select *Create»Local Variable*. A local variable icon for the object appears on the block diagram as shown in Figure 4.23.

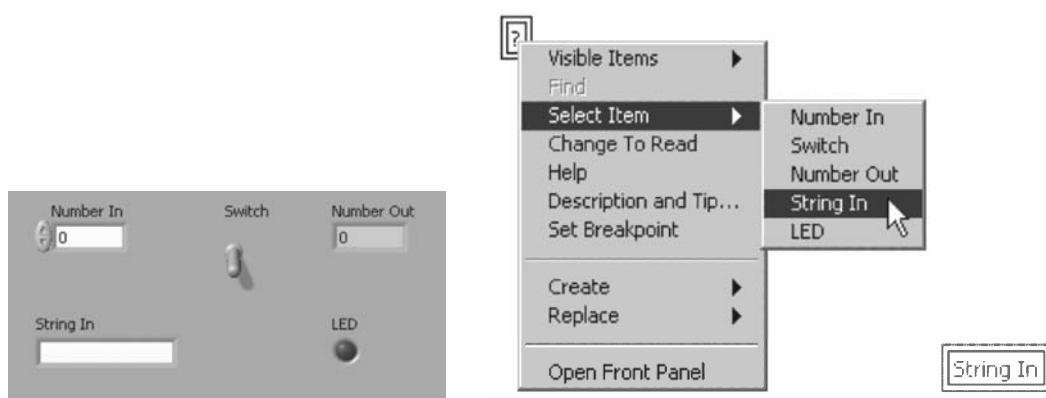


**Figure 4.22** Local variable.



**Figure 4.23** Creating a local variable from object terminal.

Another way is to select the *Local Variable* from the *Structures* palette. Create the front panel and select a local variable from the *Functions* palette and place it on the block diagram. The local variable node, shown as follows, is not yet associated with a control or indicator. To associate a local variable with a control or indicator, right-click the local variable node and select *Select Item* from the shortcut menu. The expanded shortcut menu lists all the front panel objects that have owned labels as shown in Figure 4.24. LabVIEW uses owned labels to associate local variables with front panel objects, so label the front panel controls and indicators with descriptive owned labels. Owned label becomes variable name and select whether you want to read or write to the local variable.



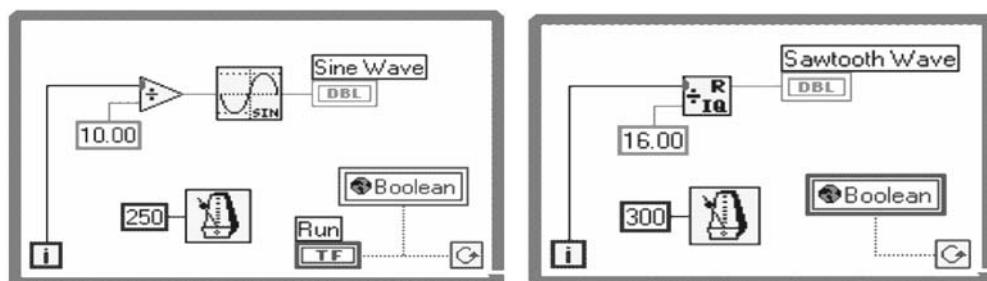
**Figure 4.24** Creating a local variable from *Structures* palette.

After you create a local or global variable, you can read data from a variable or write data to it. By default, a new variable receives data. This kind of variable works as an indicator and is a write local or global. When you write new data to the local or global variable, the associated front panel control or indicator updates to the new data. You also can configure a variable to behave as a data source, or a read local or global. Right-click the variable and select *Change To Read* from the shortcut menu to configure the variable to behave as a control. When this node executes, the VI reads the data in the associated front panel control or indicator. To change the variable to receive data from the block diagram rather than provide data, right-click the variable and select *Change To Write* from the shortcut menu. On the block diagram, you can distinguish read locals or globals from write locals or globals the same way you distinguish controls from indicators. A read local or global has a thick border similar to a control. A write local or global has a thin border similar to an indicator. When you write to a local variable, you update its corresponding front panel object and when you read from a local variable, you read the current value of its corresponding front panel object. Initialize local and global variables before reading them.

#### 4.11 GLOBAL VARIABLES

Global variables are built-in LabVIEW objects. You can use variables to access and pass data among several VIs that run simultaneously. A local variable shares data within a VI; a global variable also shares data, but it shares data with multiple VIs. For example, suppose you have two VIs running simultaneously. Each VI contains a While Loop and writes data points to a waveform chart. The first VI contains a Boolean control to terminate both VIs. You can use a global variable to terminate both loops with a single Boolean control as shown in Figure 4.25. If both loops were on a single block diagram within the same VI, you could use a local variable to terminate the loops.

When you create a global variable, LabVIEW automatically creates a special global VI, which has a front panel but no block diagram. Add controls and indicators to the front panel of the global VI to define the data types of the global variables. Select a global variable as shown in Figure 4.25 from the *Functions* palette and place it on the block diagram. Double-click the global variable node to display the front panel of the global VI. Place controls and indicators on this front panel the same way you do on a standard front panel. LabVIEW uses owned labels to identify global variables, so label the front panel controls and indicators with descriptive owned labels.



**Figure 4.25** Global variable.

Figure 4.26 shows a global variable front panel with a numeric, a string, and a cluster containing a digital and a Boolean control. The toolbar does not show the *Run*, *Stop* or related buttons as a normal front panel.



**Figure 4.26** Global variable front panel.

After you finish placing objects on the global VI front panel, save it and return to the block diagram of the original VI. You then must select which object in the global VI that you want to access. Right-click the global variable node and select a front panel object from the *Select Item* shortcut menu. The shortcut menu lists all the front panel objects that have owned labels. You also can use the operating tool or labeling tool to click the local variable node and select the front panel object from the shortcut menu. If you want to use this global variable in other VIs, select *Functions»All Functions»Select a VI*. By default, the global variable is associated with the first front panel object with an owned label that you placed in the global VI. Right-click the global variable node you placed on the block diagram and select a front panel object from the *Select Item* shortcut menu to associate the global variable with the data from another front panel object.

The two common problems that globals will cause in an application are race conditions and performance problems. These problems are pretty easy to diagnose and relatively easy to fix once you understand them. A race condition occurs when more than one piece of code needs to update a global and there is no way of ensuring that they take turns and leave the global in a valid state. The solution is to control the access and making the code take turns. The performance problem is that it is also possible to use globals in ways that greatly slow down your VI's execution. This is very different from a race condition in that the program executes correctly, but it takes longer than it should to complete. There is a very good solution that will actually make the code easier to read at the same time it restores the performance. To speed this up, make a subVI that takes the parameters for indexing the global and returns the data. Local and global variables are advanced LabVIEW

concepts. They are inherently not part of the LabVIEW dataflow execution model. Block diagrams can become difficult to read when you use local and global variables, so you should use them carefully.

## SUMMARY

---

- Use structures on the block diagram to repeat blocks of code and to execute code conditionally or in a specific order.
- The While Loop executes the subdiagram until the conditional terminal receives a specific Boolean value. By default, the While Loop executes its subdiagram until the conditional terminal receives a TRUE value.
- The For Loop executes a subdiagram a set number of times.
- You create loops by using the cursor to drag a selection rectangle around the section of the block diagram you want to repeat or by dragging and dropping block diagram objects inside the loop.
- The *Wait Until Next ms Multiple* function makes sure that each iteration occurs at certain intervals. Use this function to add timing to loops.
- The *Wait (ms)* function waits a set amount of time.
- Coercion dots appear where LabVIEW coerces a numeric representation of one terminal to match the numeric representation of another terminal.
- Use shift registers on For Loops and While Loops to transfer values from one loop iteration to the next.
- Create a shift register by right-clicking the left or right border of a loop and selecting *Add Shift Register* from the shortcut menu.
- To configure stacked shift register to remember values from multiple previous iterations and carry over values to the next iteration, right-click the left terminal and select *Add Element* from the shortcut menu.
- The feedback node stores data when the loop completes iteration, sends that value to the next iteration of the loop and transfers any data type.
- Use the feedback node to avoid unnecessarily long wires.
- Local variables allow data to be passed between parallel loops.
- Global variables allow data to be passed between VIs.

## MISCELLANEOUS SOLVED PROBLEMS

---

**Problem 4.1** Create a VI to find the factorial of the given number using For Loop and Shift Registers.

**Solution** The front panel has the number and its factorial, while the block diagram contains the codes to solve the problem as shown in Figures P4.1(a) and P4.1(b).

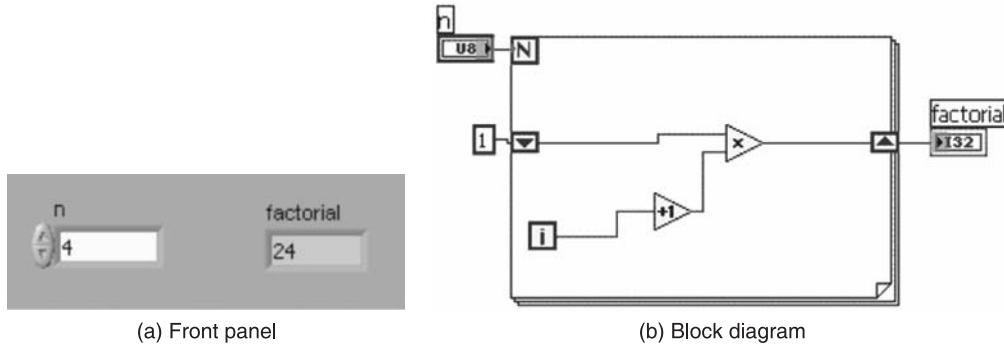


Figure P4.1

**Problem 4.2** Create a VI to find the sum of first  $n$  natural numbers using a While Loop with a feedback node.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P4.2(a) and P4.2(b). Given a number  $n$ , the sum of first  $n$  natural numbers is obtained when the program is run.

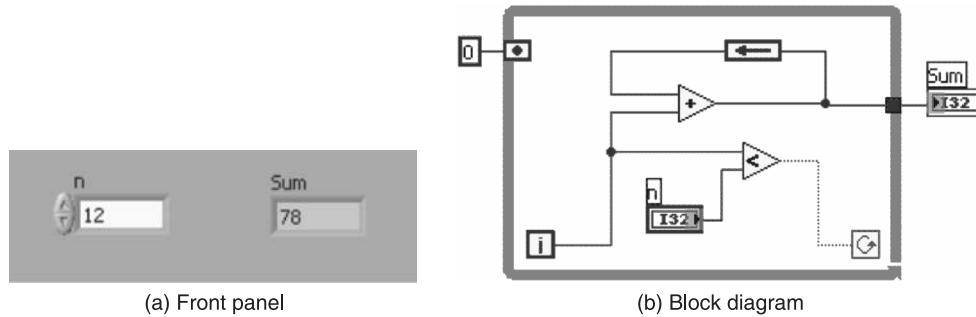


Figure P4.2

**Problem 4.3** Create a VI to change the state of the Boolean indicator  $n$  times between TRUE and FALSE.

**Solution** Build the front panel and the block diagram as shown in Figures P4.3(a) and P4.3(b) to solve the problem.

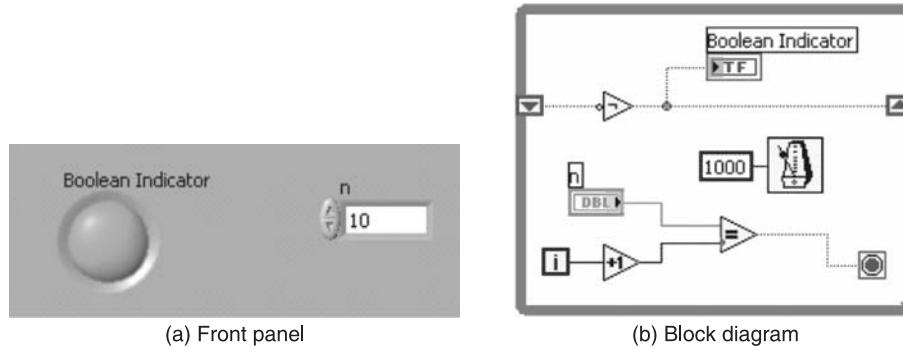


Figure P4.3

**Problem 4.4** Create a VI to find the sum of first 10 natural numbers using a For Loop.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P4.4(a) and P4.4(b).

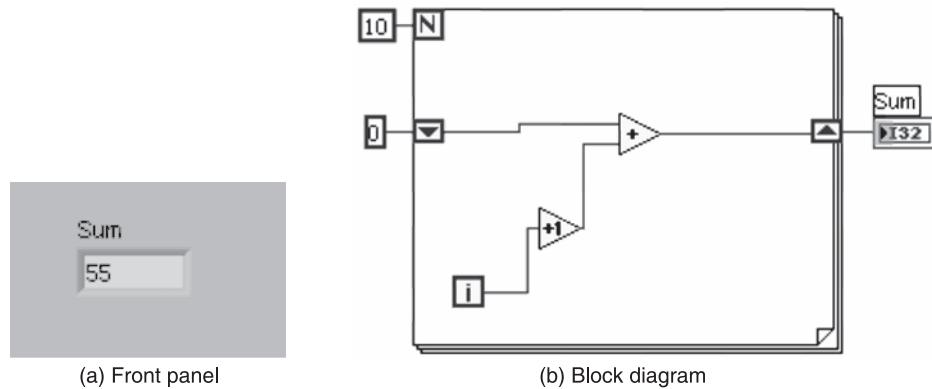


Figure P4.4

**Problem 4.5** Create a VI which converts a decimal number to a binary number using For Loops.

**Solution** The front panel and the block diagram to convert a decimal number to a binary number are shown in Figures P4.5(a) and P4.5(b).

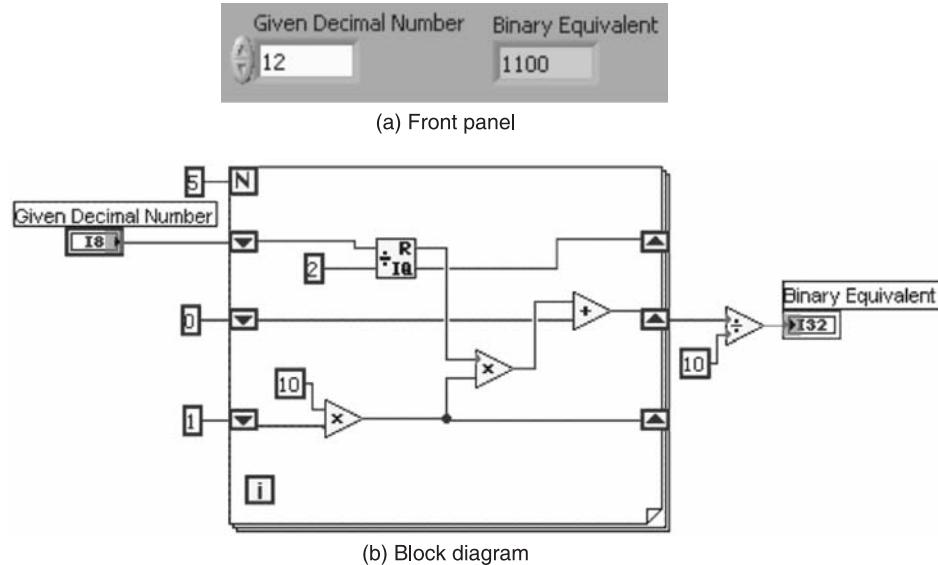


Figure P4.5

**Problem 4.6** Create a VI to find the factorial of a given number using a While Loop.

**Solution** The front panel and the block diagram to find the factorial are shown in Figures P4.6(a) and P4.6(b).

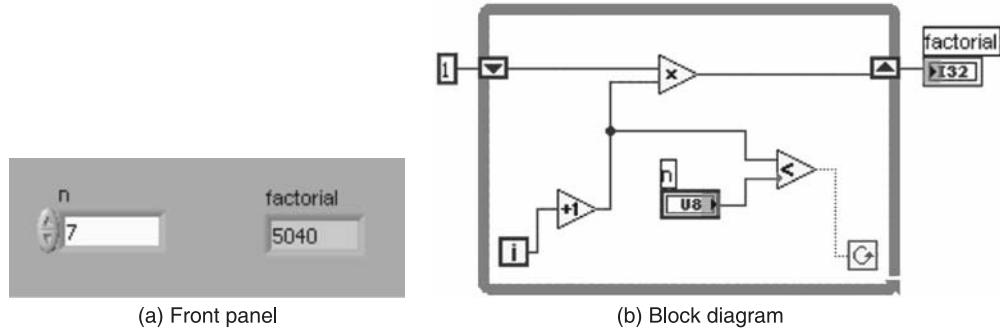


Figure P4.6

**Problem 4.7** Create a VI to find  ${}^nC_r$  and  ${}^nPr$  of a given number using a For Loop.

**Solution** The front panel and the block diagram to find  ${}^nC_r$  and  ${}^nPr$  are shown in Figures P4.7(a) and P4.7(b).

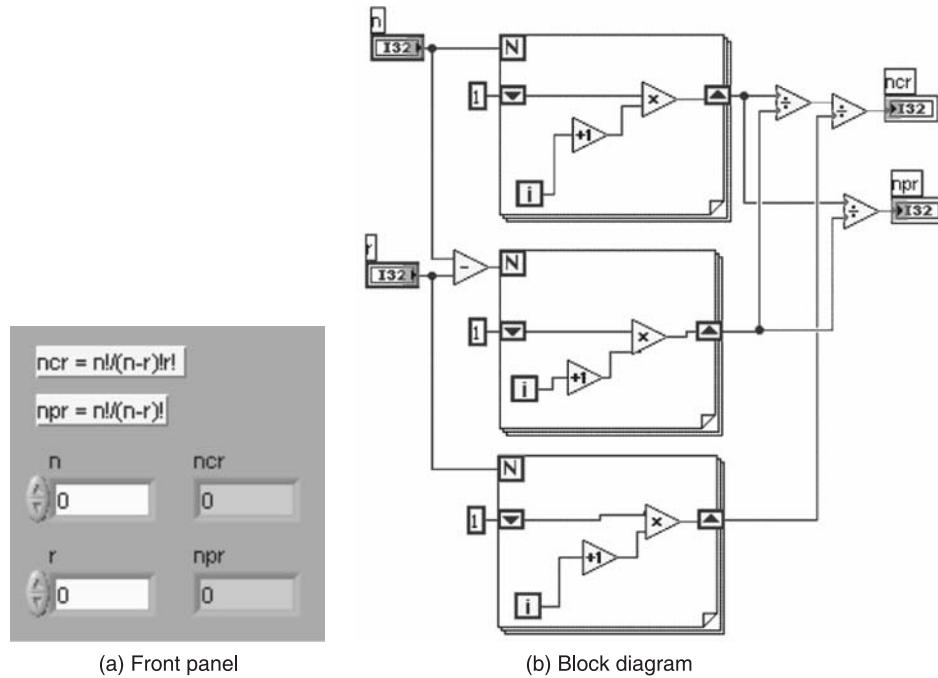


Figure P4.7

**Problem 4.8** Create a VI to animate a bird flying. Use the Picture Ring control to insert and display pictures to be animated.

**Solution** The front panel and the block diagram to solve an animate-a-bird-flying problem are shown in Figures P4.8(a) and P4.8(b).

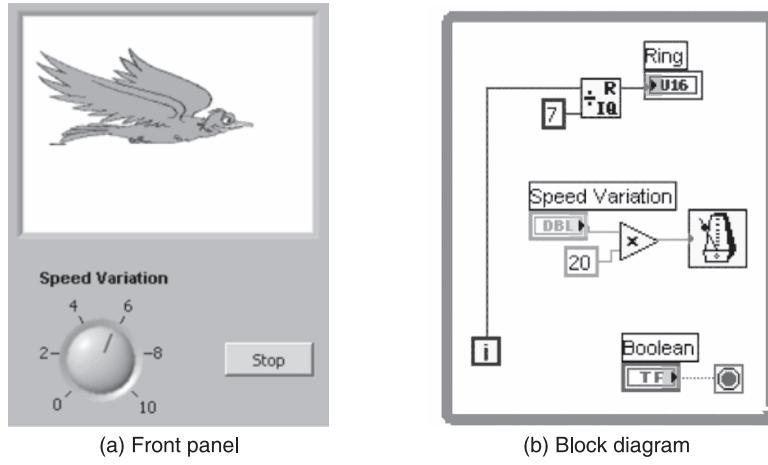


Figure P4.8

**Problem 4.9** Create a global VI (Global1.vi) which consists of a knob and a stop button. Create another VI (plot.vi) consisting of a waveform chart. Update the values of Global1.vi's knob and plot.vi's slide in the waveform chart. Press the stop button of the Global1.vi to stop both the VIs.

**Solution** The front panel is shown in Figures P4.9(a) and P4.9(b) and the block diagram to solve the global variable problem are shown in Figure P4.9(c).

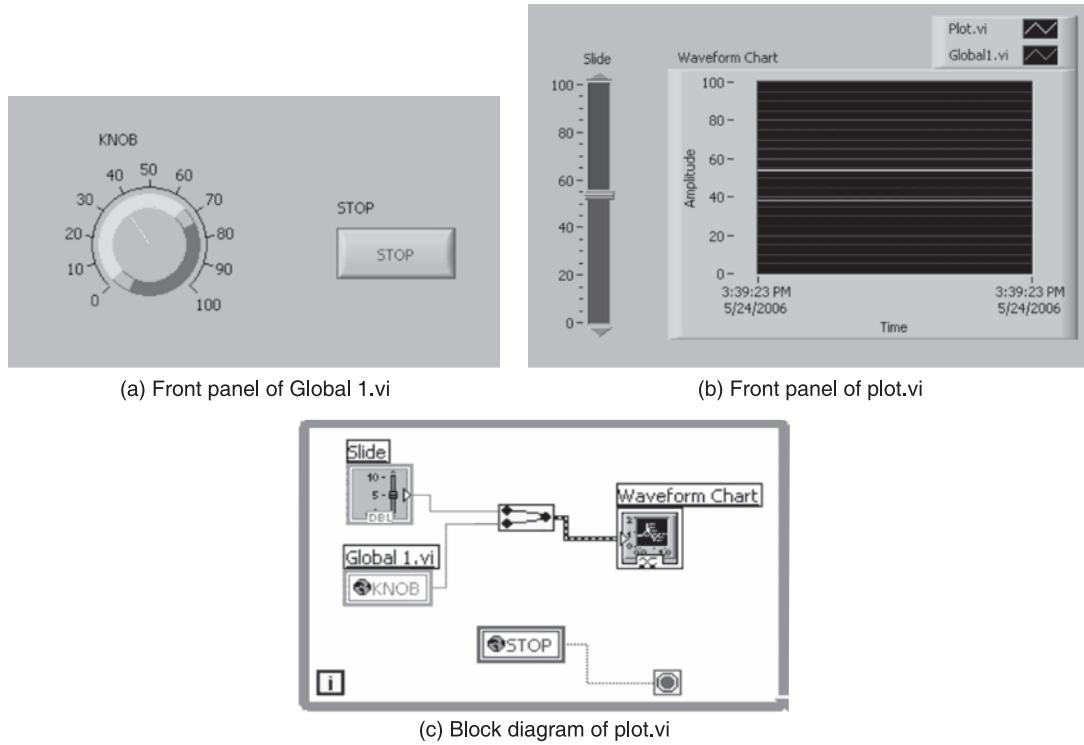
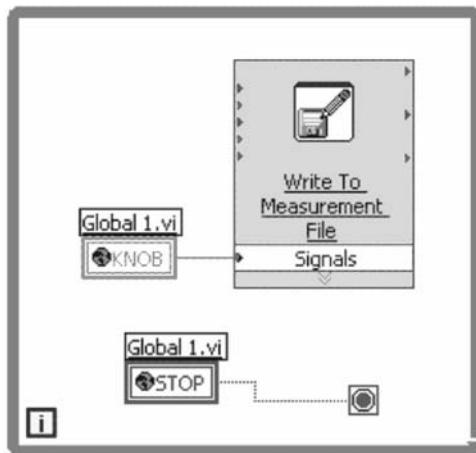


Figure P4.9

**Problem 4.10** Using the Global1.vi created in Problem 4.9, write the variation of knob values in a file using Write to Measurement File.vi. Stop the VI using the stop button of Global1.vi.

**Solution** The block diagram to solve the problem is shown in Figure P4.10.



**Figure P4.10** Block diagram.

**Problem 4.11** Create a global VI (Global2.vi) with a waveform chart and a stop button. Update the waveform chart using a random number generator from another VI (chart.vi). Stop the VIs by pressing the stop button of any one of the VIs.

**Solution** The front panel is shown in Figures P4.11(a) and P4.11(b), and the block diagram to solve the problem is shown in Figure P4.11(c).



(a) Front panel of Global2.vi

**Figure P4.11 (Contd.)**

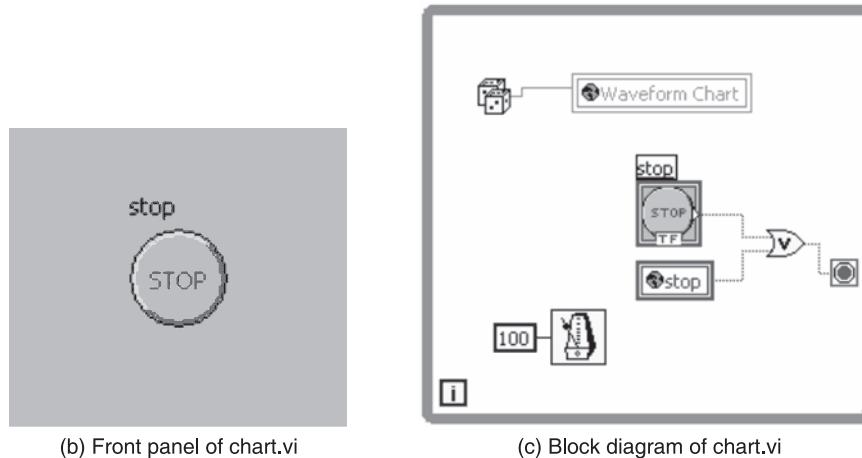


Figure P4.11

## REVIEW QUESTIONS

1. What is looping in LabVIEW? State the advantages of using loops.
2. What are the different types of loops used in LabVIEW?
3. What is a For Loop? Under what circumstances are For Loops used?
4. What is a While Loop? Under what circumstances are While Loops used?
5. How does a While Loop vary from a For Loop?
6. What is the difference between using conditional terminals, *Stop if True* and *Continue if True* in a While Loop?
7. What is a tunnel? How is it used in loops?
8. Why are shift registers and feedback nodes used in loops?
9. What is the difference between a shift register and a feedback node?
10. What are stacked shift registers? When are they used?
11. Describe the need for initializing shift registers and feedback nodes.
12. What is the difference between connecting the feedback node before and after branching?
13. What are the functions available in LabVIEW to control the timing of loops?
14. How can you control the speed at which the loop executes?
15. What is a coercion dot?
16. What are the advantages and disadvantages of local and global variables?

## EXERCISES

1. Find the sum and average of the given numbers.
2. Program to display the numbers between 0 to 10 as well as 10 to 0 simultaneously.

3. Program to display a name 27 times using a For Loop.
4. Program to find the sum of first 100 natural numbers.
5. Program to display the sum and average of the given numbers using a While Loop.
6. Program to display the numbers from 1 to 100 in steps of 3.
7. Write a program in LabVIEW to find the square of the numbers from 1 to 100 using (a) a For Loop and (b) a While Loop.
8. Write a program to display the numbers and its cube from 1 to 10 using (a) a For Loop and (b) a While Loop.
9. Write a program in LabVIEW to print the number, the square and the cubes of only even numbers from 0 to 100.
10. Write a program in LabVIEW that finds the factorial of the given numbers using (a) a For Loop and (b) a While Loop.
11. Write a program in LabVIEW that finds whether the given number is a prime number or not using (a) a For Loop and (b) a While Loop.
12. Write a program in LabVIEW to find the sum of the following series using (a) a For Loop and (b) a While Loop:
  - (i)  $1 + 2 + 3 + \dots + n$
  - (ii)  $1 - (1/1!) + (2 / 2!) - (3 / 3!)$
  - (iii)  $x + (x^2/2!) + (x^4/4!) + \dots + (x^n/n!)$
  - (iv)  $x - (x^3/3!) + (x^5/5!) + \dots + (x^n/n!)$
13. Write a program in LabVIEW to solve a quadratic equation to check all possibilities of their coefficients ( $x^2 + bx + c = 0$ ).
14. Write a program in LabVIEW to generate a Fibonacci series of  $n$  numbers where  $n$  is defined by the programmer. Example for the Fibonacci series is: 1 1 2 3 5 8 13 21 34 and so on.
15. Write a program in LabVIEW to read a positive number  $n$  and to generate the following number series using (a) a For Loop and (b) a While Loop:
  - (i) 1, 2, 3, 4, ...,  $n$
  - (ii) 0, 2, 4, 6, ...,  $n$
  - (iii) 1, 3, 5, 7, ...,  $n$
  - (iv)  $1, 2^2, 3^2, 4^2, \dots, n^2$
  - (v)  $1, 2^3, 3^3, 4^3, \dots, n^3$



# ARRAYS

---

## 5.1 INTRODUCTION

A group of homogeneous elements of a specific data type is known as an *array*, one of the simplest data structures. Arrays hold a sequence of data elements, usually of the same size and same data type placed in contiguous memory locations that can be individually referenced. Hence arrays are essentially a way to store many values under the same name. Individual elements are accessed by their position in the array. The position is given by an index, which is also called a *subscript*. The index usually uses a consecutive range of integers. Some arrays are multi-dimensional, but generally, one- and two-dimensional arrays are the most common.

You can consider using arrays when you work with a collection of similar data and when you perform repetitive computations. Arrays are ideal for storing data you collect from waveforms, graphs, data generated in loops or data collected from multiple sensors. LabVIEW like other conventional structured languages uses arrays and data structures (data structures include clusters and type definitions which are discussed in later chapters) for this purpose.

## 5.2 ARRAYS IN LabVIEW

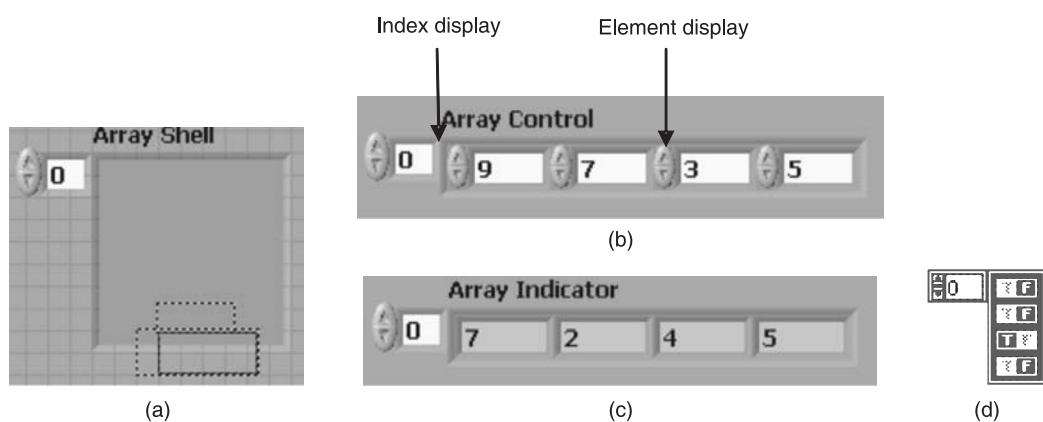
In LabVIEW arrays group data elements of the same type. They are analogous to arrays in traditional languages. An array consists of elements and dimensions. Elements are the data that make up an array. A dimension is the length, height or depth of an array. An array can have one or more dimensions and as many as  $(2^{31}) - 1$  elements per dimension, memory permitting.

You can build arrays of numeric, Boolean, path, string and cluster data types. You cannot create arrays of arrays. However, you can use a multidimensional array or an array of clusters where each cluster contains one or more arrays. Also, you cannot create an array of subpanel controls, tab controls, .NET controls, ActiveX controls, charts or multiplot XY graphs.

Array elements are ordered. To locate a particular element in an array requires one index per dimension. For example, if you want to locate a particular element in a two-dimensional array, you need both row index and column index. In LabVIEW, indexes let us navigate through an array and retrieve elements, rows and columns from an array on the block diagram. The index is zero-based, which means it is in the range 0 to  $n - 1$ , where  $n$  is the number of elements in the array with the first element at index 0 and the last one at index  $(n - 1)$ . For example, if you create an array of 10 elements, the index ranges from 0 to 9. The fourth element has an index of 3.

### 5.3 CREATING ONE-DIMENSIONAL ARRAY CONTROLS, INDICATORS AND CONSTANTS

Create an array control or indicator on the front panel by placing an array shell on the front panel as shown in Figure 5.1(a), and dragging a data object or element, which can be a numeric, Boolean, string, path, refnum, or cluster control or indicator, into the array shell. The array shell automatically resizes to accommodate the new object. The array shell can be selected from *Controls>>Modern>>Arrays, Matrix & Clusters* palette. The array elements must be controls or indicators. You must insert an object in the array shell before you use the array on the block diagram. Otherwise, the array terminal appears black with an empty bracket. After placing an element in the array shell, you can expand the array either horizontally or vertically to see more number of elements. Once a data type is assigned to the array shell, the block diagram takes the color and lettering (in [ ] brackets) of the data type. For example, if the data type is a numeric indicator, the color will be orange with [DBL] written inside the terminal. Arrays can be identified easily by their thicker (or double) wires. Figure 5.1(b) shows an array of numeric controls and the array has four elements. Figure 5.1(c) shows an array of numeric indicators.



**Figure 5.1** (a) Array shell placed on the front panel, (b) Array of numeric controls, (c) Array of numeric indicators and (d) Array of Boolean constants.

The index ranges from 0 to 3. The first element in the array (9) is at index 0, the second element (7) is at index 1, the third element (3) is at index 2 and the fourth element (5) is at index 3. In an array the element selected in the index display always refers to the element shown in the upper-left corner of the element display. In Figure 5.2, the element (9) at index 0 is not shown in the array, because index 1 is selected in the index display. If you want to see the element 9 again, the index value in the index display should be changed to 0.



**Figure 5.2** Array of numeric control with index 1 selected in the index display.

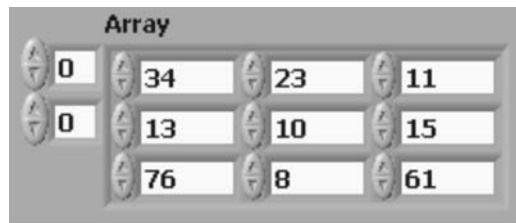
You can create an array constant on the block diagram by combining an array with a valid constant, which can be a numeric, Boolean, string, path, refnum, or cluster constant. The element cannot be another array. Following are the steps for creating an array constant in the block diagram.

1. Select an array constant from *Functions>>Programming>>Arrays* palette; place the array shell on the block diagram. The array shell appears, with an index display on the left, an empty element display on the right, and an optional label.
2. Place a constant in the array shell. For example, a numeric constant in the array shell.
3. The array shell automatically resizes to accommodate the object you place in the array shell. When an object is placed in the array shell, the data type of the array constant is defined.
4. An alternative method of creating array constant is to copy or drag an existing array on the front panel to the block diagram to create a constant of the same data type.

You can use an array constant to store constant data or as a basis for comparison with another array. Array constants are also useful for passing data into a subVI. Figure 5.1(d) shows an array of Boolean constants.

#### 5.4 CREATING TWO-DIMENSIONAL ARRAYS

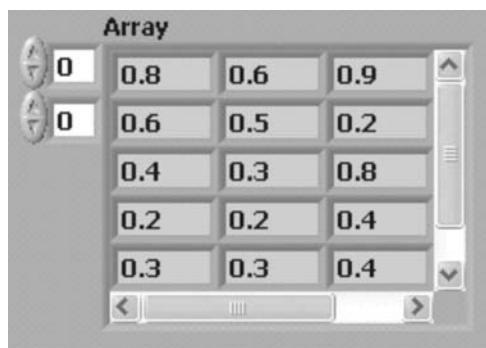
A two-dimensional array is analogous to a spreadsheet or table. A two-dimensional array stores elements in a grid. It requires a column index and a row index to locate an element, both of which are zero-based. Two-dimensional arrays are very commonly used in data acquisition applications. For example, when waveforms from several channels are read from a data acquisition (DAQ) board, the data is stored in a two-dimensional array where each column in the array corresponds to data from one channel. To create a two-dimensional array on the front panel, right-click the index display of the array and select *Add Dimension* from the shortcut menu. You also can use the Positioning tool to resize the index display to have one more dimension. Figure 5.3 shows a two-dimensional array.



**Figure 5.3** Two-dimensional array.

## 5.5 CREATING MULTIDIMENSIONAL ARRAYS

To create a multidimensional array on the front panel, right-click the index display of the array and select *Add Dimension* from the shortcut menu. You also can use the Positioning tool to resize the index display until you have as many dimensions as you want. To delete dimensions one at a time, right-click the index display and select *Remove Dimension* from the shortcut menu. You can also resize the index display to delete dimensions. Use the Positioning tool to resize the array to show more than one row or column at a time. To display a particular element on the front panel, either type the index number in the index display or use the arrows on the index display to navigate to that number. You can also use the scroll bars of an array to navigate to a particular element. Right-click the array and select *Visible Items»Vertical Scrollbar* or *Visible Items»Horizontal Scrollbar* from the shortcut menu to display scroll bars for the array as shown in Figure 5.4.



**Figure 5.4** Array with horizontal and vertical scroll bars.

## 5.6 INITIALIZING ARRAYS

You can initialize an array, or leave it uninitialized. When an array is initialized, you can define the number of elements in each dimension and the contents of each element. An uninitialized array has a dimension but no elements. Figure 5.5(a) shows an uninitialized two-dimensional array control with all the elements are dimmed indicating that the array is uninitialized. Figure 5.5(b) shows an array of two rows and two columns.

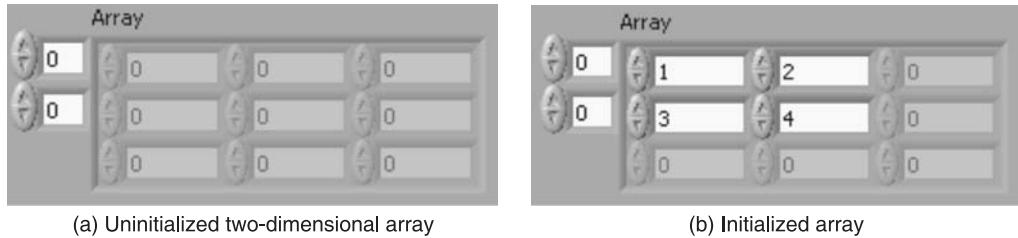


Figure 5.5

## 5.7 DELETING ELEMENTS, ROWS, COLUMNS AND PAGES WITHIN ARRAYS

You can delete an element within a one-dimensional array and a row or column within a two-dimensional array. To delete an element in a one-dimensional array, right-click the array element on the front panel and select *Data Operations»Delete Element*. To delete a row or column in a two-dimensional array, right-click the array row or column on the front panel and select *Data Operations»Delete Row* or *Delete Column*. You can also programmatically delete elements, rows, columns and pages within arrays using the *Delete From Array* function.

You can delete an element, row, column or page within an array programmatically. What you can delete depends on how many dimensions the array has. For example, you can delete a row or a column from an array of two or more dimensions. You can delete a page from an array of three or more dimensions.

Complete the following steps to delete elements, rows, columns or pages in an array.

**Step 1:** Place the *Delete From Array* function on the block diagram.

**Step 2:** Wire an array of any dimension to the *n*-dim array input of the *Delete From Array* function. The function automatically resizes based on the dimensions of the array.

**Step 3:** Determine which operation you want to perform from Table 5.1 and complete the associated steps.

The index input specifies from which element, row, column, or page you want to start deleting, with 0 being the first. The length input specifies the number of elements, rows, columns, or pages you want to delete.

**Step 4:** Run the VI.

**TABLE 5.1** Operations for deleting elements, rows, columns of pages

Array wired to n-dimension array	Deleting	Complete these steps
One-dimensional array	Element(s)	Wire a value 1–n to <b>length</b> . Wire a value 0–n to <b>index</b> .
Two-dimensional array	Row(s)	Wire a value 1–n to <b>length</b> . Wire a value 0–n to <b>index (row)</b> .
	Column(s)	Wire a value 1–n to <b>length</b> . Wire a value 0–n to <b>index (column)</b> .
Three-dimensional multidimensional array	Page(s)	Wire a value 1–n to <b>length</b> . Wire a value 0–n to <b>index (page)</b> .

## 5.8 INSERTING ELEMENTS, ROWS, COLUMNS AND PAGES INTO ARRAYS

You can insert an element into a one-dimensional array and a row or column into a two-dimensional array. To add an element to a one-dimensional array, right-click the array on the front panel and select *Data Operations»Insert Element Before*. To add a row or column to a two-dimensional array, right-click the array on the front panel and select *Data Operations»Insert Row Before* or *Insert Column Before*. You also can programmatically insert elements, rows, column, and pages into arrays using the *Insert Into Array* function.

You can insert an element, row, column or page into an array programmatically. What you can insert depends on how many dimensions the array has. For example, you can insert a row or a column into an array of two or more dimensions, but you cannot insert an element. You can insert a page into an array of three or more dimensions, but you cannot insert an element, row, or column. Complete the following steps to insert elements, rows, columns or pages in an array.

1. Place an *Insert Into Array* function on the block diagram.
2. Wire an array of any dimension to the *n*-dim array input of the *Insert into Array* function. The function automatically resizes based on the dimensions of the array.
3. Determine which operation you want to perform from Table 5.2 and complete the associated steps.
4. The index input specifies the element, row, column or page where you want to insert the element or array, with 0 being the first. Elements and arrays are added before the value you wire to index. Resize the *Insert Into Array* function to insert another element(s) row(s), column(s), or page(s) in an array and repeat steps 2 and 3.
5. Run the VI.

If the dimension size of the array you insert is smaller than the dimension size of the array you wired to the *n*-dim array input of the *Insert Into Array* function, the function pads the array you insert with default data, such as zeros in the case of an array of numeric values. If the dimension size of the array you insert is larger than the dimension size of the array you wired to the *n*-dim array input of the *Insert Into Array* function, the function crops the array from the end.

**TABLE 5.2** Operations for inserting elements, rows, columns and pages

Array wired to n-dimension array	Inserting	Complete these steps
One-dimensional array	Element	Wire a value 0–n to <b>index</b> . Wire a value to <b>new element/subarray</b> .
	One-dimensional array to row or column	Wire a value 0–n to <b>index</b> . Wire a one dimensional array to <b>new element/subarray</b> .

(Contd.)

**TABLE 5.2** (*Contd.*)

<b>Array wired to n-dimension array</b>	<b>Inserting</b>	<b>Complete these steps</b>
Two-dimensional array	Row	Wire a value 0–n to <b>index (row)</b> . Wire a one dimensional array to <b>new element subarray</b> .
	Column	Wire a value 0–n to <b>index (column)</b> . Wire a one dimensional array to <b>new element/ subarray</b> .
	Two-dimensional array (rows and columns)	Wire a value 0–n to <b>index (row)</b> or <b>index (column)</b> . Wire a two dimensional array to <b>new element/ sub array</b> .
Three-dimensional-nD array	Page	Wire a value 0–n to <b>index (page)</b> . Wire a two dimensional array to <b>new element/ sub array</b> .
	Three dimensional array (pages)	Wire a value 0–n to <b>index (page)</b> . Wire a three dimensional array to <b>new element/sub array</b> .

## 5.9 REPLACING ELEMENTS, ROWS, COLUMNS, AND PAGES WITHIN ARRAYS

You can replace an element, row, column or page in an array. What you can replace depends on how many dimensions the array has. For example, in an array of two or more dimensions, you can replace a row or a column with a one-dimensional array. In an array of three or more dimensions, you can replace a page with a two-dimensional array. Replacing is done using the *Replace Array Subset* function. Complete the following steps to replace elements, rows, columns or pages in an array.

**Step 1:** Place the *Replace Array Subset* function on the block diagram.

**Step 2:** Wire an array of any dimension to the *n*-dimension array input of the Replace Array Subset function. The function automatically resizes based on the dimensions of the array.

**Step 3:** Determine which operation you want to perform from Table 5.3 and complete the associated steps.

The index input specifies which element, row, column or page to replace, with 0 being the first. The new element/subarray input specifies the value you want to replace an element, or the array you want to replace a row, column or page.

**Step 4:** Resize the *Replace Array Subset* function to replace another element, row, column or page within an array and repeat steps 2 and 3.

**Step 5:** Run the VI.

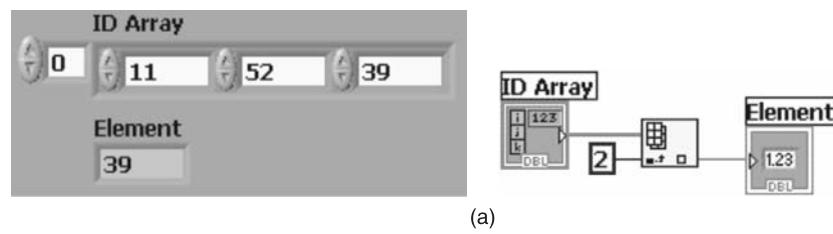
**TABLE 5.3** Operations for replacing elements, rows, columns and pages

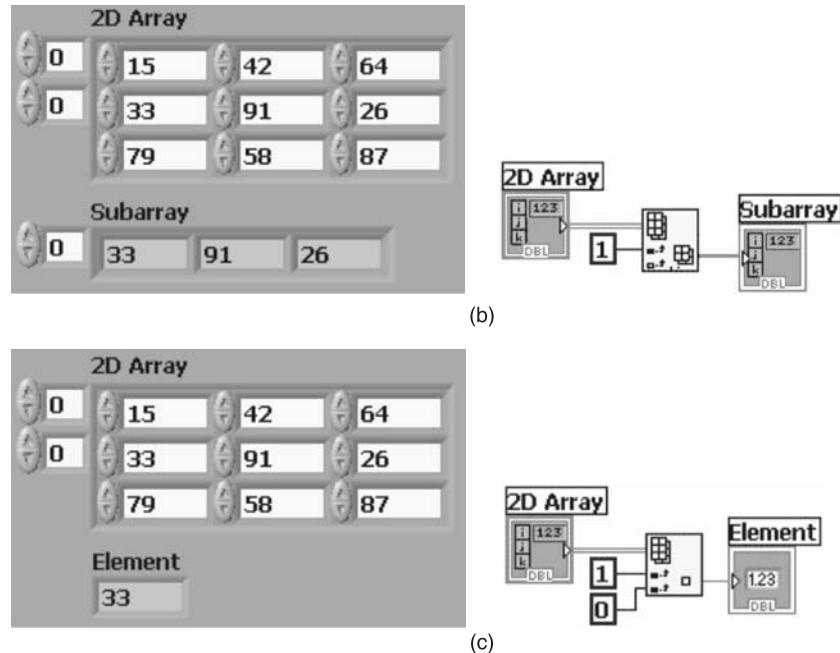
Array wired to $n$ -dimension array	Replacing	Complete these steps
One-dimensional array	Element	Wire a value 0– $n$ to <b>index</b> . Wire a value to <b>new element/subarray</b> .
Two-dimensional array	Row	Wire a value 0– $n$ to <b>index (row)</b> . Wire a one dimensional array to <b>new element/subarray</b> .
	Column	Wire a value 0– $n$ to <b>index (column)</b> . Wire a one dimensional array to <b>new element/subarray</b> .
	Element	Wire a value 0– $n$ to <b>index (row)</b> . Wire a value 0– $n$ to <b>index (column)</b> . Wire a value to <b>new element/subarray</b> .
Three-dimensional- $n$ D array	Page	Wire a value 0– $n$ to <b>index (page)</b> . Wire a two dimensional array to <b>new element/subarray</b> .
	Row	Wire a value 0– $n$ to <b>index (page)</b> . Wire a value 0– $n$ to <b>index (row)</b> . Wire a one dimensional array to <b>new element/subarray</b> .
	Column	Wire a value 0– $n$ to <b>index (page)</b> . Wire a value 0– $n$ to <b>index (column)</b> . Wire a one dimensional array to <b>new element/subarray</b> .
	Element	Wire a value 0– $n$ to <b>index (page)</b> . Wire a value 0– $n$ to <b>index (row)</b> . Wire a value 0– $n$ to <b>index (column)</b> . Wire a value to <b>new element/subarray</b> .

## 5.10 ARRAY FUNCTIONS

Array functions are used to create and manipulate arrays. You can perform common array operations such as extracting individual data elements from an array, inserting, deleting, or replacing data elements in an array or splitting arrays using array functions.

Array functions including *Index Array*, *Replace Array Subset*, *Insert Into Array*, *Delete From Array*, and *Array Subset* automatically resize to match the dimensions of the input array you wire. For example, if you wire a one-dimensional array to one of these functions, the function shows a single index input. If you wire a two-dimensional array to the same function, it shows two index inputs—one for the row index and one for the column index.

**Figure 5.6** (Contd.)



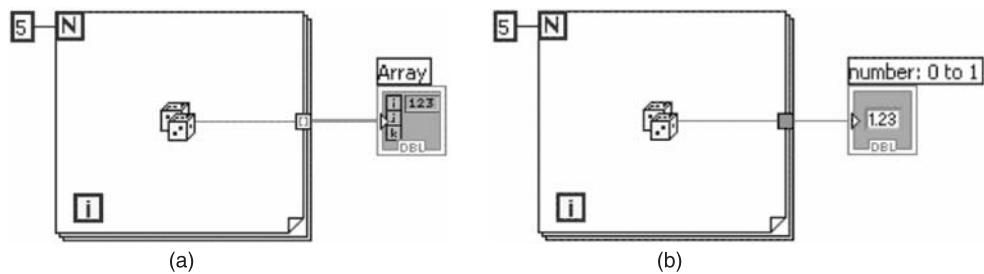
**Figure 5.6** (a) Index array function with one-dimensional array as input, (b) index array function with two-dimensional array as input and with one index (row index) and (c) index array function with two-dimensional array as input and with two indices.

In the above example shown in Figure 5.6, the *Index Array* function is used. In Figure 5.6(a), input to the *Index Array* function is a one-dimensional array. By providing the index value in the output, you get the array element corresponding to the index value. When connecting a two-dimensional array as input, the *Index Array* function automatically resizes to get two index inputs, one for row index and other for column index. In Figure 5.6(b), input to the *Index Array* function is a two-dimensional array. The row index is provided. At the output, you get a one-dimensional array which is a row of the two dimensional input array. The row output is based on the index input provided. In Figure 5.6(c), input to the *Index Array* function is again a two-dimensional array. Here both row index and column index are given. The output is an element which is based on the row and column indices provided.

You can access more than one element, or subarray (row, column, or page) with these functions by using the Positioning tool to manually resize the function. When you expand one of these functions, the functions expand in increments determined by the dimensions of the array wired to the function. If you wire a one-dimensional array to one of these functions, the function expands by a single index input. If you wire a two-dimensional array to the same function, the function expands by two index inputs—one for the row and one for the column. The index inputs you wire determine the shape of the subarray you want to access or modify. For example, if the input to an *Index Array* function is a two-dimensional array and you wire only the row input, you extract a complete row (one-dimensional) of the array. If you wire only the column input, you extract a complete column (one-dimensional) of the array. If you wire the row input and the column input, you extract a single element of the array. Each input group is independent and can access any portion of any dimension of the array.

## 5.11 AUTO INDEXING

For Loops and While Loops can index and accumulate arrays at their boundaries. This is known as *auto-indexing*. If you wire an array to a For Loop or While Loop input tunnel, you can read and process every element in that array by enabling auto-indexing. When you auto-index an array output tunnel, the output array receives a new element from every iteration of the loop. The wire from the output tunnel to the array indicator becomes thicker as it changes to an array at the loop border, and the output tunnel contains square brackets representing an array as shown in Figure 5.7.

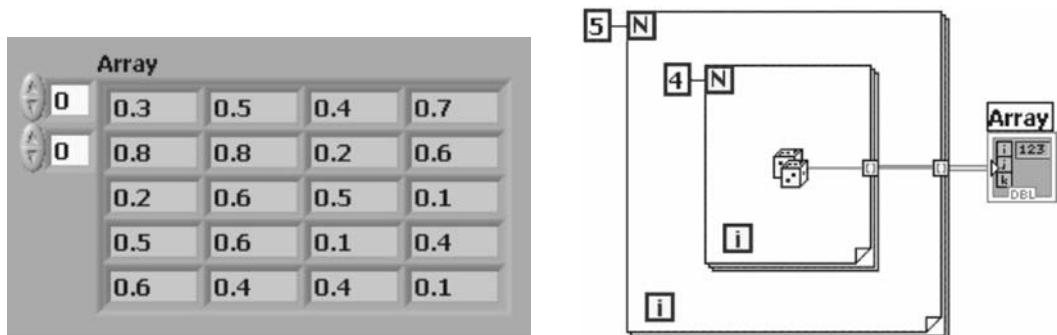


**Figure 5.7** (a) For Loop with enabled auto indexing and (b) For Loop with disabled auto-indexing.

Disable auto-indexing by right-clicking the tunnel and selecting *Disable Indexing* from the shortcut menu. Disable auto-indexing if you need only the last value passed to the tunnel. Because you can use For Loops to process arrays by one element at a time, LabVIEW enables auto-indexing by default for every array you wire to a For Loop and for each output tunnel that is created. Auto-indexing for While Loops is disabled by default. To enable auto-indexing, right-click a tunnel and select *Enable Indexing* from the shortcut menu.

## 5.12 CREATING TWO-DIMENSIONAL ARRAYS USING LOOPS

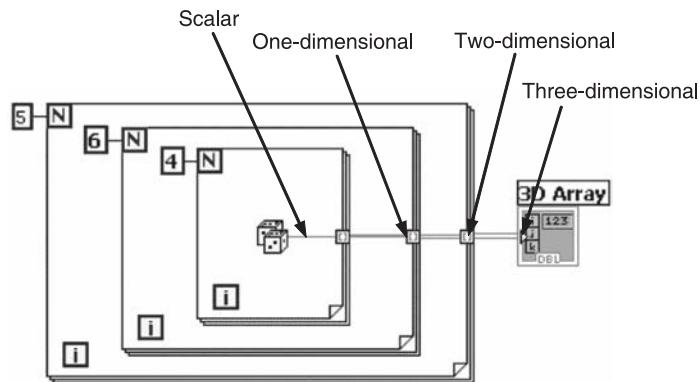
A two-dimensional array can also be generated easily using two nested For Loops, i.e. by placing one For Loop inside the other as shown in Figure 5.8. In this case the outer loop will correspond to the rows of the array and the inner one to the columns. In the example shown in Figure 5.8, the two-dimensional array generated consists of five rows and four columns.



**Figure 5.8** Two-dimensional array using two nested For Loops.

### 5.13 IDENTIFICATION OF DATA STRUCTURE (SCALAR AND ARRAY) USING WIRES

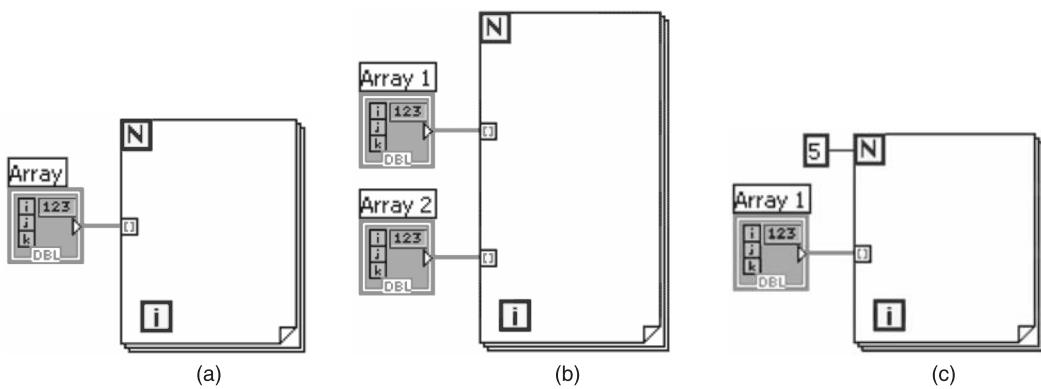
Scalar values are indicated by thin wire, one-dimensional arrays are indicated by a thick wire and two-dimensional arrays are indicated by two wires with color appropriate to the data type. Arrays of larger dimensions are also indicated by two wires, but with a greater spacing. A three-dimensional (3D) array, a  $(i, j, k)$  will have ‘ $i$ ’ indicating the page, ‘ $j$ ’ the row and ‘ $k$ ’ the column. One can consider the array to be like a pack of cards, with the page indicating the card in the deck. All indices start from zero. Figure 5.9 shows the wires corresponding to a scalar as well as one-dimensional, two-dimensional and three-dimensional arrays.



**Figure 5.9** Difference in wires corresponding to scalar, one-, two- and three-dimensional arrays.

### 5.14 USING AUTO-INDEXING TO SET THE FOR LOOP COUNT

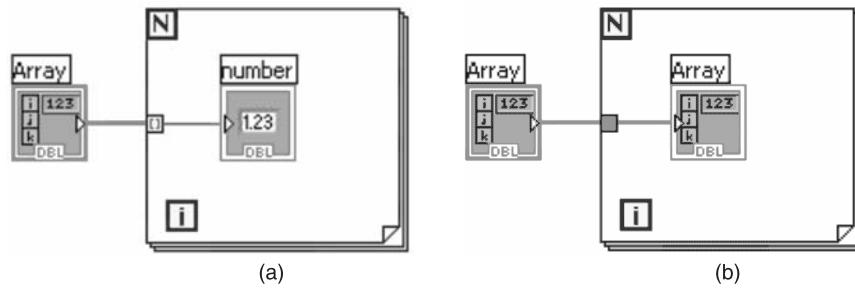
If you enable auto-indexing on an array wired to a For Loop input terminal, LabVIEW sets the count terminal to the array size so you do not need to wire the count terminal. In Figure 5.10(a), the For Loop executes a number of times equal to the number of elements in the array.



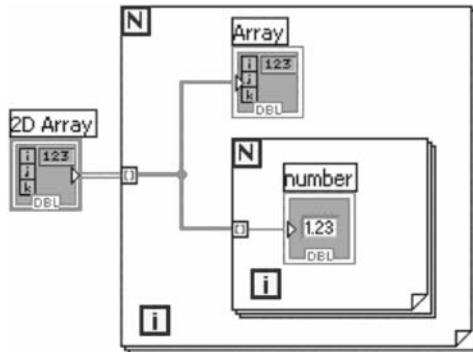
**Figure 5.10** (a) Setting the For Loop count with enabled auto-indexing for one tunnel, (b) Setting the For Loop count with enabled auto-indexing for more than one tunnel and (c) Setting the For Loop count with enabled auto-indexing for one tunnel and count terminal.

Normally, if the count terminal of the For Loop is not wired, the run arrow is broken. However, in this case the run arrow is not broken. If you enable auto-indexing for more than one tunnel (as shown in Figure 5.10(b)) or if you wire the count terminal (as shown in Figure 5.10(c)), the count changes to the smaller of the two. For example, if you wire an array with 10 elements to a For Loop input tunnel and you set the count terminal to 15, the loop executes only 10 times.

Auto-indexing can be used on input arrays. For calculations to be performed on each element of array, use auto-indexing as shown in Figure 5.11(a). To pass the entire array into a loop, disable auto-indexing as shown in Figure 5.11 (b). If auto-indexing is enabled for a two-dimensional array as shown in Figure 5.12, the output from the tunnel will be a one-dimensional array. For example, if you wire a two-dimensional array of three rows, the For Loop executes for three times and the tunnel output will be three one-dimensional arrays with one row per iteration. To get a particular element from the one-dimensional array, another For Loop has to be placed inside the previous loop as shown in Figure 5.12.



**Figure 5.11** (a) Enabled auto-indexing for input array and (b) Disabled auto-indexing for input array.



**Figure 5.12** Enabled auto-indexing for two-dimensional input array.

## 5.15 MATRIX OPERATIONS WITH ARRAYS

Use the matrix data type to make modeling of math problems easier. Matrices group rows or columns of real or complex scalar data for some math operations, such as linear algebra operations. A two-dimensional array is similar to a matrix.

### 5.15.1 Converting an Array to a Matrix

You can convert a one-dimensional or two-dimensional array of double-precision or complex numbers to a matrix and continue to use array functions to manipulate the matrix values. Complete the following steps to convert an array to a matrix.

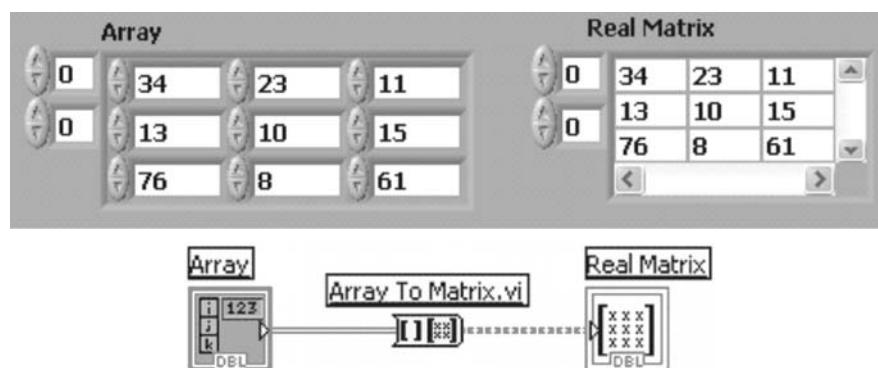
**Step 1:** Place a one-dimensional or two-dimensional array of floating-point values on the front panel.

**Step 2:** Place the *Array To Matrix* function on the block diagram.

**Step 3:** Wire the array to the *Array To Matrix* function.

**Step 4:** Right-click the *Array To Matrix* function and select *Create»Indicator* from the shortcut menu to create a matrix indicator.

**Step 5:** Run the VI. The matrix indicator displays the contents of the original array as shown in Figure 5.13.



**Figure 5.13** Front panel and block diagram of the VI to convert an array to a matrix.

If you wire a one-dimensional array to the *Array to Matrix* function, the function copies the array elements to the first column of the matrix. The function converts arrays that contain complex elements to complex matrices and converts all other arrays to real matrices. LabVIEW stores matrix elements with double-precision numerical values.

### 5.15.2 Converting a Matrix to an Array

You can convert a matrix to a two-dimensional array and use array functions to manipulate the matrix values.

Complete the following steps to convert a matrix to a two-dimensional array.

**Step 1:** Place a *Real Matrix* or *Complex Matrix* control on the front panel.

**Step 2:** Place the *Matrix To Array* function on the block diagram.

**Step 3:** Wire a matrix to the *Matrix To Array* function.

**Step 4:** Right-click the *Matrix To Array* function and select *Create»Indicator* from the shortcut menu to create a two-dimensional array indicator.

**Step 5:** Run the VI from the front panel. The array indicator displays the contents of the array as shown in Figure 5.14.

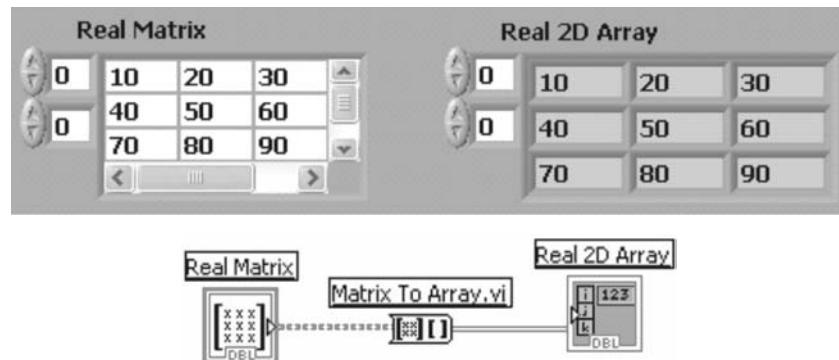


Figure 5.14 Front panel and block diagram of the VI to convert a matrix into an array.

You can store a matrix of a single row or column in a one-dimensional array. However, LabVIEW cannot determine the dimensions of a matrix until runtime. The *Matrix To Array* function returns has the same element type, real or complex, as the matrix you wired to the *Matrix To Array* function.

## 5.16 POLYMORPHISM

All LabVIEW arithmetic functions are polymorphic. This means that the inputs to these functions can be different data structures such as scalar values and arrays. By definition *polymorphism* is said to be the ability of a numeric function to adjust to input data of different data structures.

For example, the *Add* function can be used in the following ways.

1. Scalar + Scalar = Scalar addition.
2. Scalar + Array = The scalar is added to each element of array.
3. Array + Array = Each element of one array is added to the corresponding element of other array.

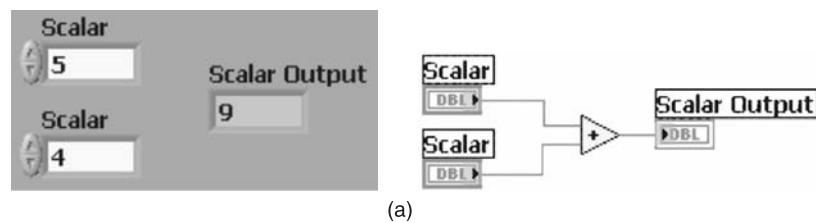
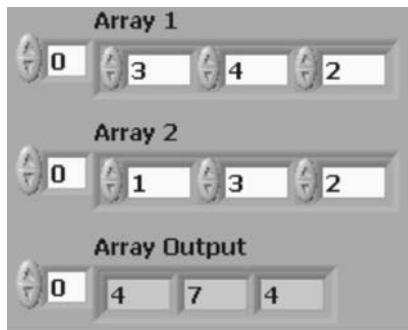
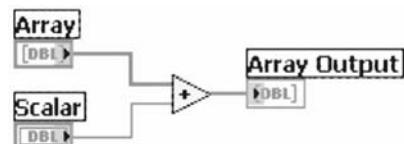


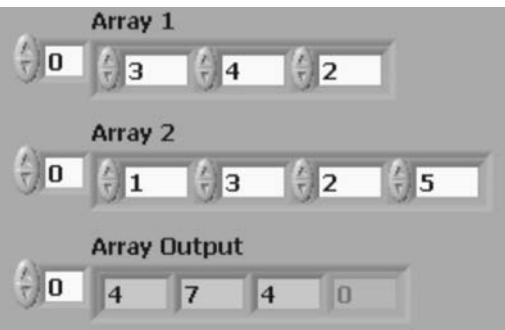
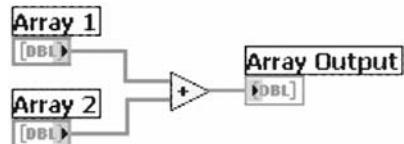
Figure 5.15 (Contd.)



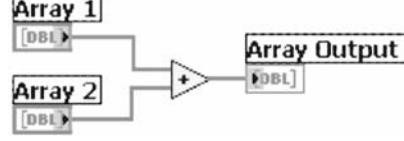
(b)



(c)



(d)



**Figure 5.15** (a) Adding two scalar values, (b) adding a scalar value with an array, (c) adding two arrays of the same size and (d) adding two arrays of different sizes.

In Figure 5.15(a), two scalar values are added which results in a scalar output. In Figure 5.15(b), an array is added with a scalar value. The function adds the scalar value to each element of the array and returns the array output. In Figure 5.15(c), two arrays are added. The function adds each element of one array to the corresponding element of the other array and returns the array output. In Figure 5.15(d), two arrays of different sizes (i.e. number of elements) are added. The function adds corresponding elements and returns the output array, with the size of the smaller input array. The first array consists of three elements and the second consists of four elements. First three elements of both the arrays are added. The fourth element of the second array

is not considered. Polymorphism does not perform matrix arithmetic when inputs are two-dimensional arrays (for example, two two-dimensional array inputs to a multiply function do element by element multiplication, not matrix multiplication).

## SUMMARY

- Arrays group data elements of the same type. You can build arrays of numeric, Boolean, path, string, waveform and cluster data types.
- The array index is zero-based, which means it is in the range 0 to  $n - 1$ , where  $n$  is the number of elements in the array.
- You must insert an object in the array shell before you use the array on the block diagram. Otherwise, the array terminal appears black with an empty bracket.
- To create an array control or indicator; select an array on the *Controls»Array & Cluster* palette, place it on the front panel, and drag a control or indicator into the array shell.
- If you wire an array to a For Loop or While Loop input tunnel, you can read and process every element in that array by enabling auto-indexing.
- Use the array functions located on the *Functions»All Functions»Array* palette to create and manipulate arrays.
- By default, LabVIEW enables auto-indexing in For Loops and disables auto-indexing in While Loops.
- Polymorphism is the ability of a function to adjust to input data of different data structures.

## MISCELLANEOUS SOLVED PROBLEMS

**Problem 5.1** Create a one-dimensional (1D) numeric array using the *Build Array* function which gets array elements from numeric controls.

**Solution** The front panel and the block diagram to create a one-dimensional (1D) numeric array are shown in Figures P5.1(a) and P5.1(b).

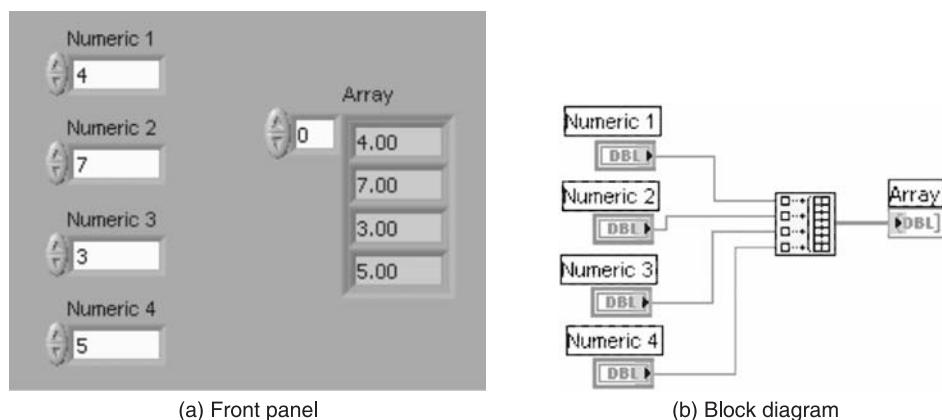


Figure P5.1

**Problem 5.2** Create a 1D numeric array from loops (For and While) using random numbers and obtain the reverse of the array.

**Solution** The front panel and the block diagram to create a 1D numeric array from loops are shown in Figures P5.2(a) and P5.2(b).

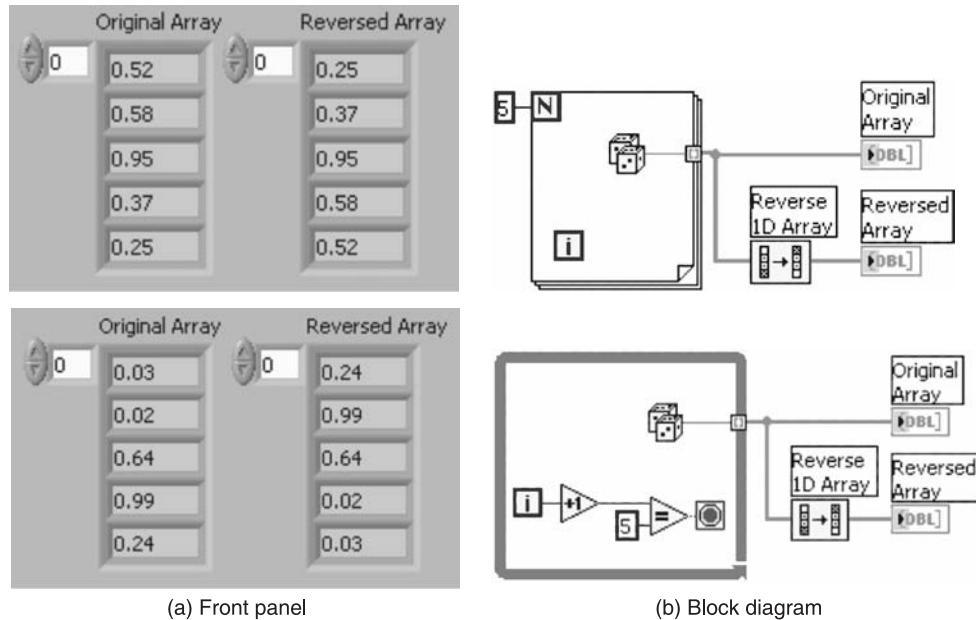


Figure P5.2

**Problem 5.3** Create a VI to find the determinant of a  $2 \times 2$  matrix which is represented in the form of a 2D array using Index array function.

**Solution** Build the front panel and the block diagram to find the determinant of a  $2 \times 2$  matrix are shown in Figures P5.3(a) and P5.3(b).

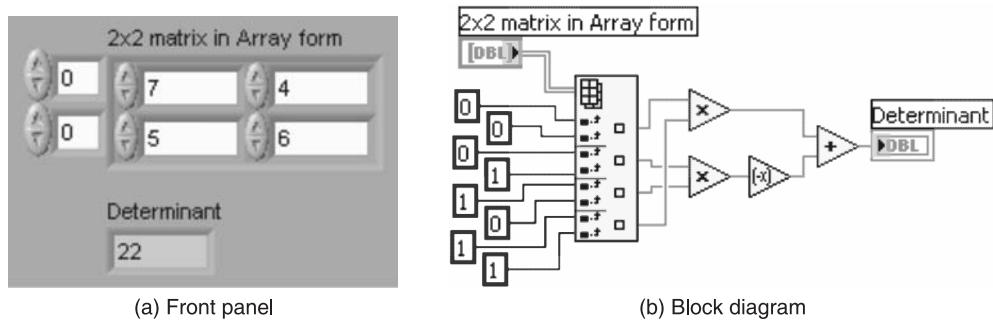
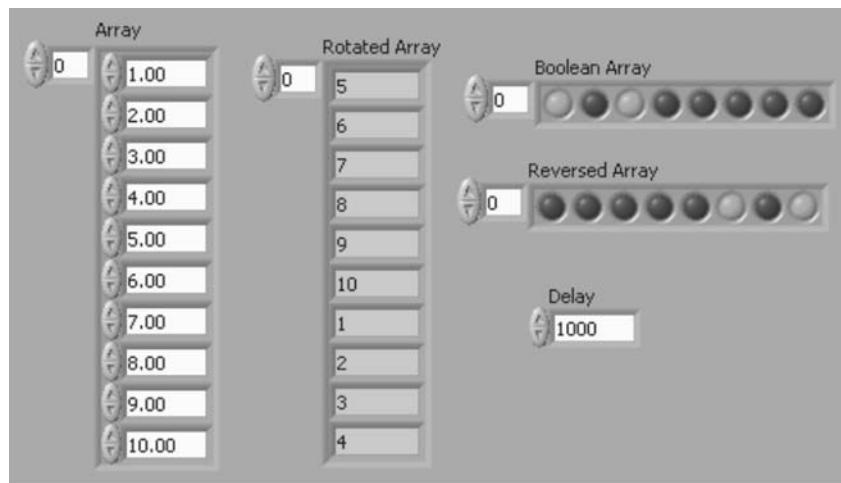


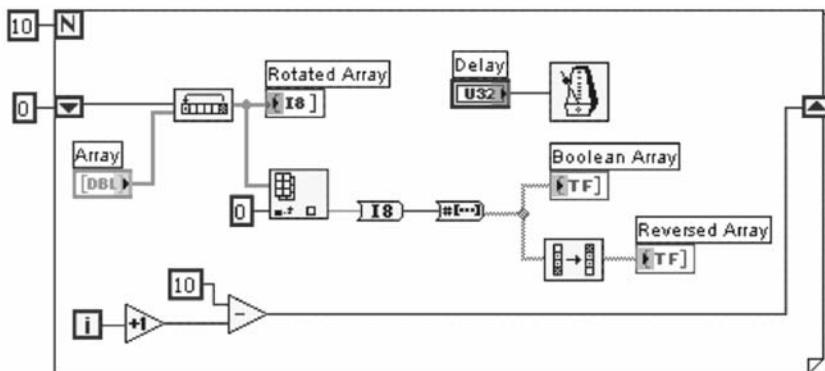
Figure P5.3

**Problem 5.4** Create a 1D numeric array which consists of ten elements and rotate it ten times. For each rotation display the equivalent binary number of the first array element in the form of a Boolean array. Also display the reversed Boolean array. Provide delay to view the rotation.

**Solution** The front panel and the block diagram to build a 1D numeric array are shown in Figures P5.4(a) and P5.4(b).



(a) Front panel



(b) Block diagram

**Figure P5.4**

**Problem 5.5** Create a 2D numeric array ( $5 \times 5$ ) containing random numbers and find its transpose.

**Solution** The front panel and the block diagram to build a 2D numeric array are shown in Figures P5.5(a) and P5.5(b).

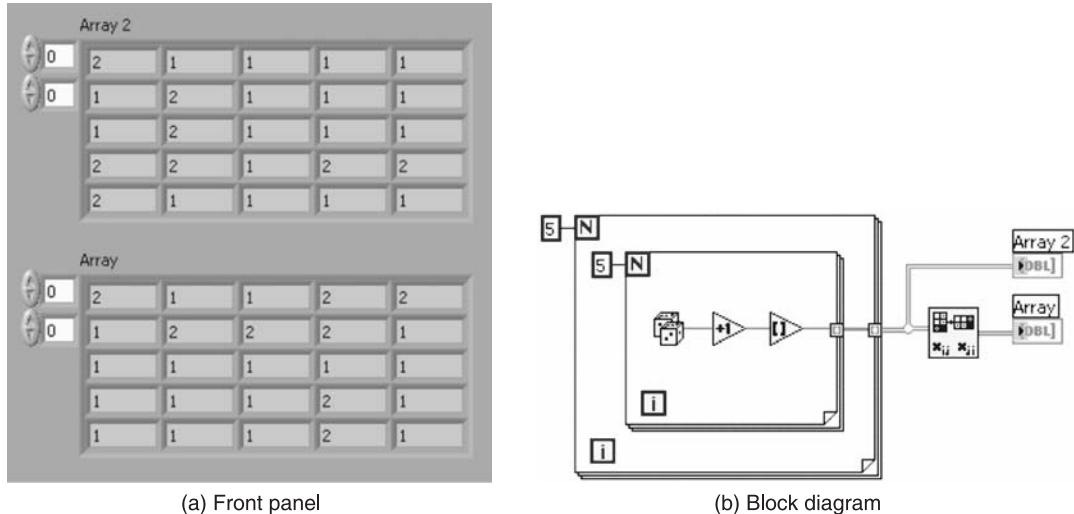


Figure P5.5

**Problem 5.6** Create two 2D numeric arrays and add them. Change the number of rows and number of columns of each array and see the result.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P5.6(a) and P5.6(b).

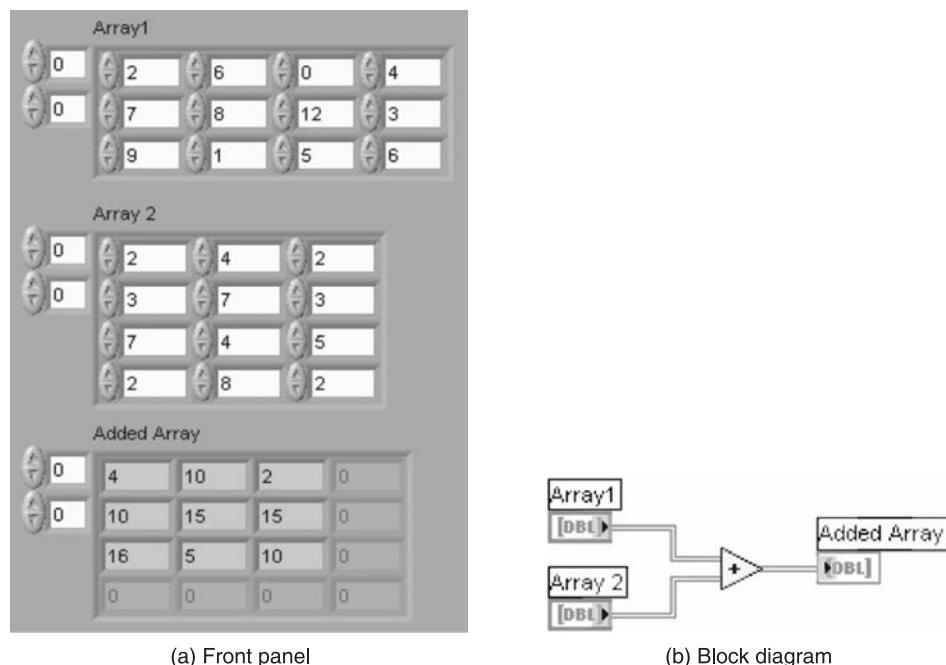
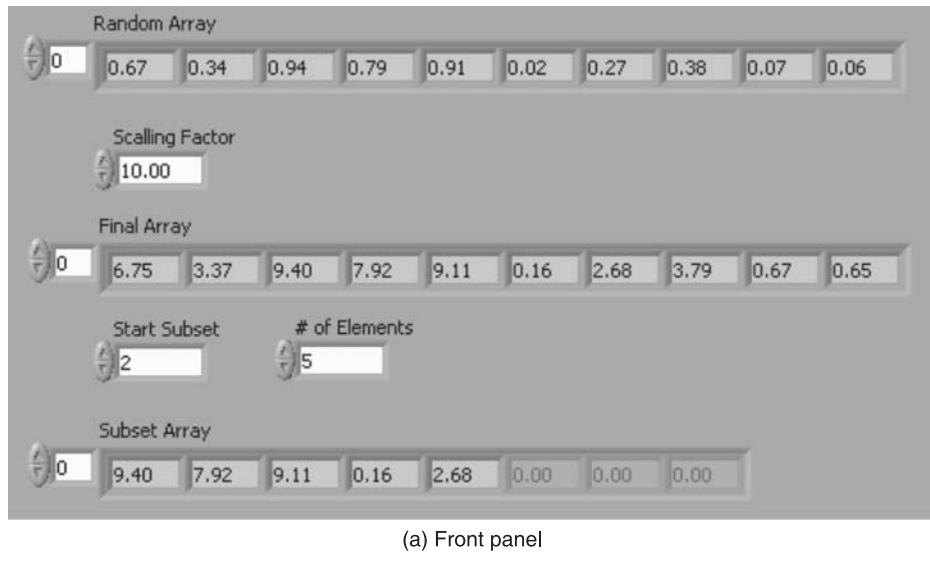


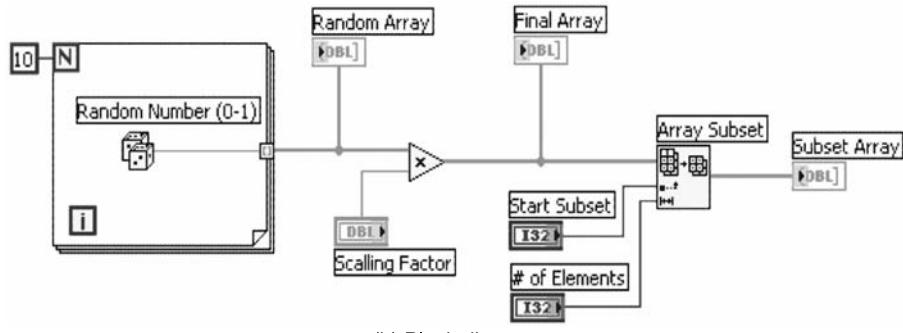
Figure P5.6

**Problem 5.7** Build a VI that generates a 1D array. Multiply the array elements by a scaling factor and find the resultant array. Create a subset array from the resultant array.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P5.7(a) and P5.7(b).



(a) Front panel



(b) Block diagram

**Figure P5.7**

**Problem 5.8** Build a VI that generates a 1D array and then multiplies pairs of elements together and outputs the resulting array. For example, the input array with values 1, 23, 10, 5, 7, 11 will result in the output array 23, 50, 77.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P5.8(a) and P5.8(b).

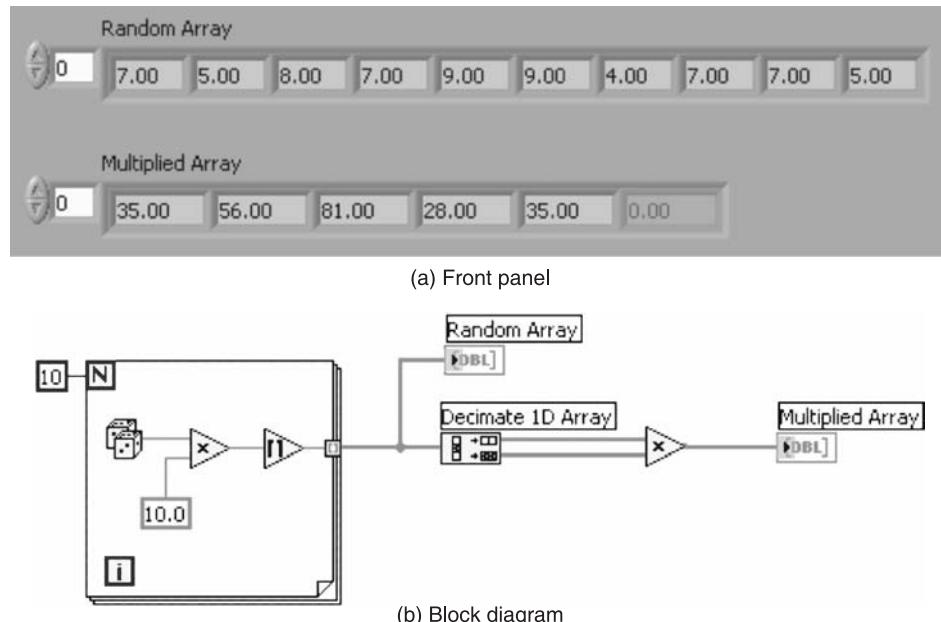


Figure P5.8

**Problem 5.9** Build a VI to find the sum and product of array elements.

**Solution** The front panel and the block diagram to find the sum and product of array elements are shown in Figures P5.9(a) and P5.9(b).

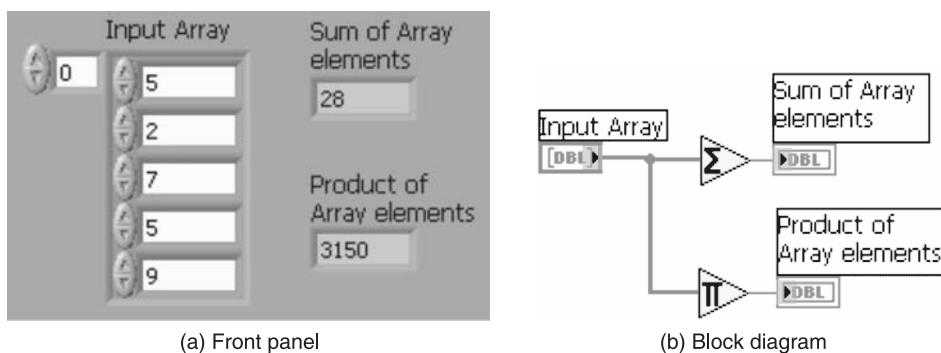


Figure P5.9

**Problem 5.10** Build a VI to solve a linear equation using matrix functions.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P5.10(a) and P5.10(b).

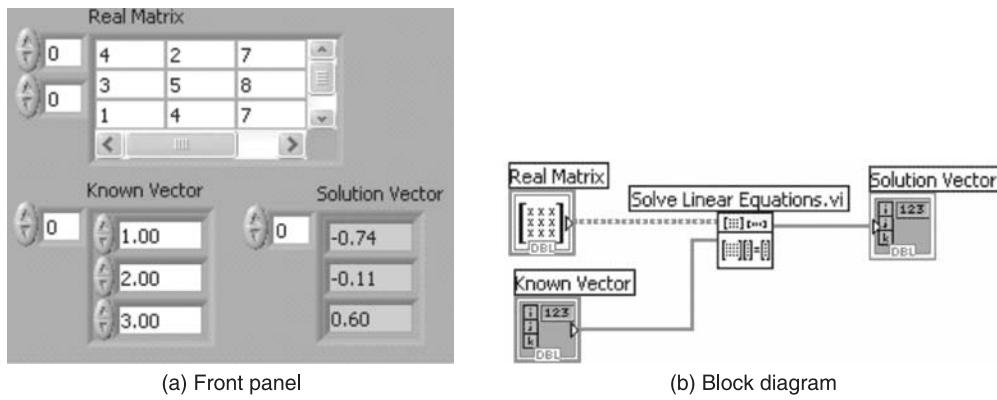


Figure P5.10

**Problem 5.11** Build a VI to find the product of two matrices using matrix function.

**Solution** Create the front panel and the block diagram to solve the problem as shown in Figures P5.11(a) and P5.11(b).

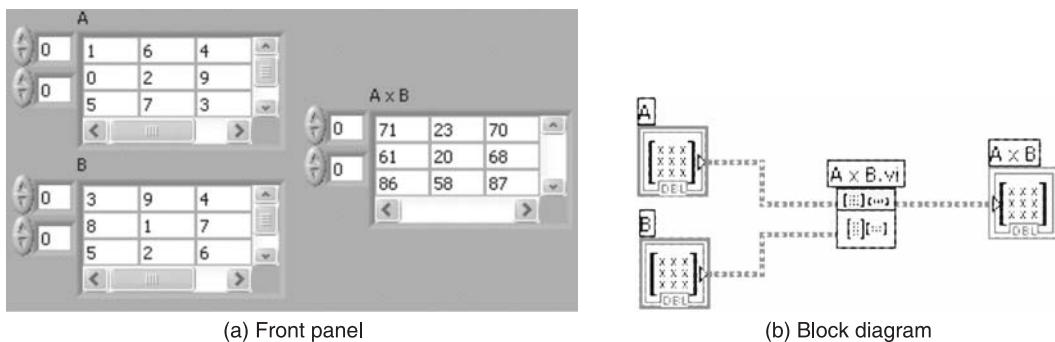


Figure P5.11

**Problem 5.12** Build a VI to find the determinant of a matrix using the matrix function.

**Solution** Create the front panel and the block diagram to find the determinant of a matrix as shown in Figures P5.12(a) and P5.12(b).



Figure P5.12

**Problem 5.13** Build a VI to find the rank of a matrix using matrix functions.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P5.13(a) and P5.13(b).



Figure P5.13

## REVIEW QUESTIONS

1. Define an array in LabVIEW. Where might it be used?
2. What is an array indexing?
3. How are the individual elements accessed and processed in an array?
4. What is array initialization?
5. What is multidimensional array? How is it different from one-dimensional array?
6. How are the individual elements of a multidimensional array accessed and processed?
7. How to create a two-dimensional array of numeric controls?
8. How can a loop create an array? Which loops can be used to do so?
9. Define auto-indexing.
10. What does “polymorphic” mean?

## EXERCISES

1. Create a 1D Boolean array and obtain the reverse of the array.
2. Create a 1D numeric array and check whether the array elements are odd or even. In the output array display 0s and 1s for odd numbers and even numbers respectively.
3. Create a VI to generate random numbers between 0–50, 50–100, 0–100 and put them into separate arrays.
4. Build a VI that generates a 1D array of random numbers and sort the array in ascending and descending order and find the maximum and minimum value of the array elements and also find the size of the array.
5. Build a VI that generates two 1D arrays and create another array which consists all the elements of the first two arrays.

- 
6. Create a VI to read a set of numbers from the keyboard and sort them in ascending order.
  7. Create a VI to read a set of numbers from the keyboard and find the sum of the array elements.
  8. Create a VI to pick up the largest number from any  $5 \times 5$  matrix.
  9. Create a VI to obtain the transpose of a  $4 \times 4$  matrix. The transpose of a matrix is obtained by exchanging the elements of each row with the elements of the corresponding column.
  10. Create a VI to read the integer number and find out the sum of all the digits until it comes to a single digit using an array.

For example:

$$n = 1346$$

$$\text{Sum} = 1 + 3 + 4 + 6 = 14$$

$$\text{Sum} = 1 + 4 = 5$$

11. Create a VI to read a number  $n$ , and print digit by digit as a series of words. For example, the number 746 would be printed as “Seven Four Six”.
12. Create a VI to read a set of numbers up to  $n$ , where the programmer defines  $n$  and print the contents of the array in reverse order.

For example: for  $n = 4$ , the set be

26      46      41      123

would be as

123      41      46      26

13. Create a VI to read a set of numbers, where  $n$  is defined by the programmer and find the average of the non-negative integer numbers and also find out the deviation of the numbers.
14. Create a VI to read a set of numbers and store it on the one-dimensional array and again read a number  $d$  and check whether the number  $d$  is present in the array. If it is so, print out how many times the number  $d$  is repeated in the array.
15. Create a VI to read a set of numbers and store it on the one-dimensional array and again read a number  $n$ , and check whether it is present in the array. If it is so, print out the position of  $n$  in the array and also check whether it repeats in the array.
16. Create a VI to read a set of numbers and store it on the one-dimensional array and find out the largest and the smallest numbers. Find out the difference of the two numbers. Using the difference, find the deviation of the numbers of the array.
17. Create a VI to read a set of numbers to store it on the one-dimensional array A, copy the elements in the other array B in the back direction, find the sum of the individual elements of array A and array B, store the results in an array C, and display all the three arrays.
18. Create a VI to read a four-digit positive integer number  $n$  and generate all the possible permutations of those numbers digit by digit. For example,  $n = 7812$ , the permutations are 7821, 8721, 8712, 2871, 2817.
19. Create a VI to read two-dimensional square matrix A, and display its transpose. The transpose of the two-dimensional array is a second array containing the same elements on the first, but in which the rows become columns and vice versa.
20. Create a VI to read a two-dimensional array and find the sum of the elements in the row-wise and column-wise separately and display the sums of the rows and columns.
21. Create a VI to generate a magic square A, where the sum of elements in the row-wise and column-wise is the same.

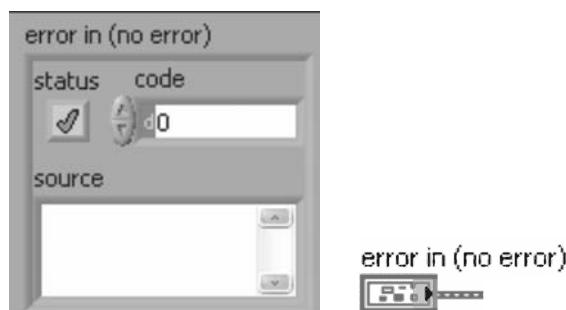


# CLUSTERS

---

## 6.1 INTRODUCTION

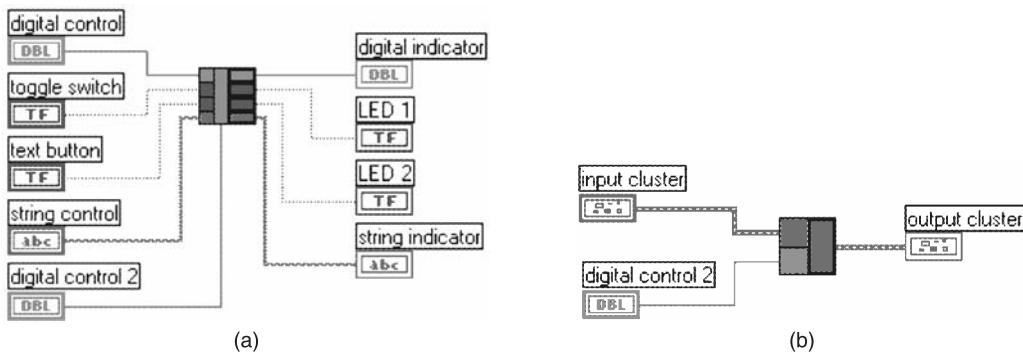
Clusters group data elements of mixed types. An example of a cluster is the LabVIEW error cluster, which combines a Boolean value, a numeric value and a string. A cluster is similar to a record or a struct in text-based programming languages. Figure 6.1 show the error cluster control and the corresponding terminal created in the block diagram. This cluster consists of a Boolean control (status), a numeric control (code) and a string control (source).



**Figure 6.1** Error cluster control.

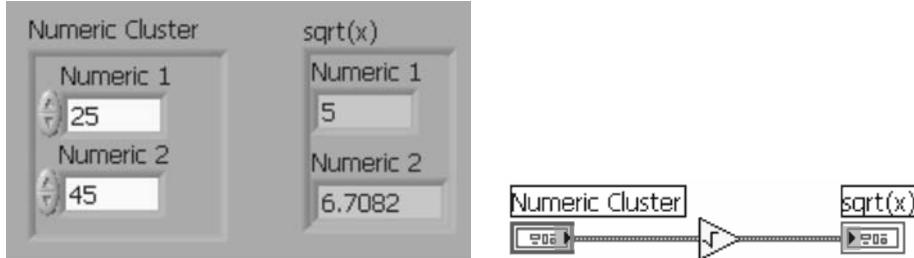
Bundling several data elements into clusters eliminates wire clutter on the block diagram. It also reduces the number of connector pane terminals that subVIs need by passing several values to one terminal. The connector pane has, at most, 28 terminals. If your front panel contains more than

28 controls and indicators that you want to pass to another VI, group some of them into a cluster and assign the cluster to a terminal on the connector pane. In Figure 6.2(a) individual values are passed to the connector pane terminals. In Figure 6.2(b) controls and indicators are grouped into clusters and connected to one control panel terminal each.



**Figure 6.2** (a) Individual values passed to subVI terminals and (b) Cluster passed to subVI terminal.

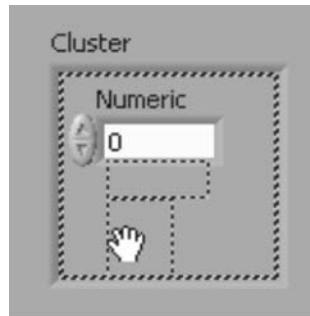
Most clusters on the block diagram have a pink wire pattern and data type terminal. Clusters of numeric values, sometimes referred to as points, have a brown wire pattern and data type terminal as shown in Figure 6.3. You can wire brown numeric clusters to numeric functions, such as *Add* or *Square Root*, to perform the same operation simultaneously on all elements of the cluster.



**Figure 6.3** Numeric cluster control, indicator and block diagram terminals.

## 6.2 CREATING CLUSTER CONTROLS AND INDICATORS

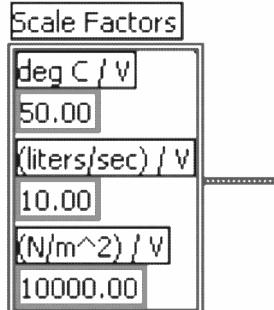
A cluster can be created by placing a cluster shell on the front panel and then placing one of the front panel objects inside the cluster as shown in Figure 6.4. Select a cluster on the *Controls>>All Controls>>Arrays & Cluster* palette, place it on the front panel and drag a data object or element, which can be a numeric, Boolean, string, path, refnum, array, or cluster control or indicator, into the cluster shell. Resize the cluster shell by dragging the cursor while you place the cluster shell on the front panel. On the block diagram the cluster wire will appear as a rope. Similar to the arrays objects can be deposited inside the cluster and all of them are either controls or indicators.



**Figure 6.4** Creation of a cluster control.

### 6.3 CREATING CLUSTER CONSTANT

To create a cluster constant on the block diagram, first select a cluster constant on the *Functions* palette. Next place the cluster shell on the block diagram, and finally place a string constant, numeric constant, or cluster constant in the cluster shell. Figure 6.5 shows a cluster constant with three numeric constants. You can use a cluster constant to store constant data or as a basis for comparison with another. If you have a cluster control or indicator on the front panel and you want to create a cluster constant containing the same elements on the block diagram, you can either drag that cluster from the front panel to the block diagram or right-click the cluster on the front panel and select *Create»Constant* from the shortcut menu. When you place objects in the constant shell, you define the data type of the cluster constant. You can also copy or drag an existing cluster control on the front panel to the block diagram to easily create a constant with the same element data types.



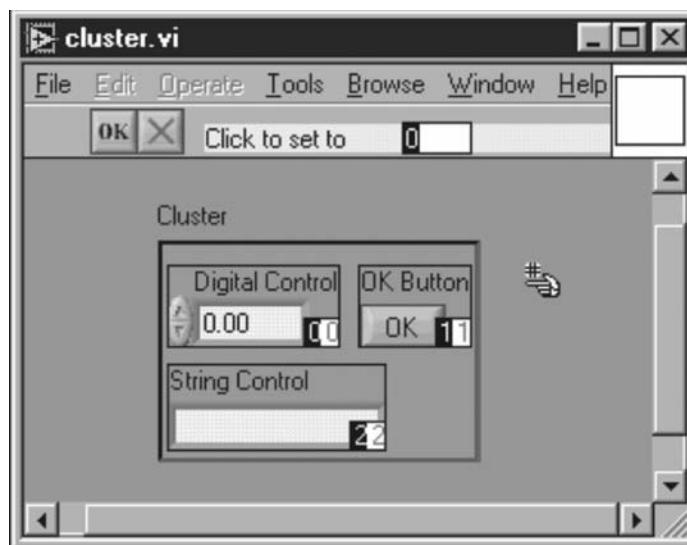
**Figure 6.5** Cluster constant.

### 6.4 ORDER OF CLUSTER ELEMENTS

Cluster elements have a logical order unrelated to their position in the shell. The first object you place in the cluster is element 0, the second is element 1, and so on. If you delete an element, the order adjusts automatically. The cluster order determines the order in which the elements appear as terminals on the *Bundle* and *Unbundle* functions on the block diagram. You can view and modify the cluster order by right-clicking the cluster border and selecting *Reorder Controls In Cluster* from the shortcut menu.

To wire clusters to each other, both clusters must have the same number of elements. Corresponding elements, determined by the cluster order, must have compatible data types. For example, if a double-precision floating-point numeric value in one cluster corresponds in the cluster order to a string in the another cluster, the wire on the block diagram appears broken and the VI does not run. If the numeric values are different representations, LabVIEW coerces them to the

same representation. Figure 6.6 shows the reordering of a cluster which contains a numeric control (*Digital Control*), a Boolean control (*OK Button*) and a string control. By clicking over the number displayed with a black background near the cluster element, we can change the order of the elements. Cluster and array elements are both ordered, you must unbundle all cluster elements at once or use the *Unbundle By Name* function to access specific cluster elements. Clusters also differ from arrays in that they are a fixed size. Like an array, a cluster is either a control or an indicator.



**Figure 6.6** Reordering cluster elements.

## 6.5 CLUSTER OPERATIONS

The main cluster operations are bundle, unbundled, bundle by name and unbundle by name. Use the cluster functions to create and manipulate clusters. For example, you can perform tasks similar to the following:

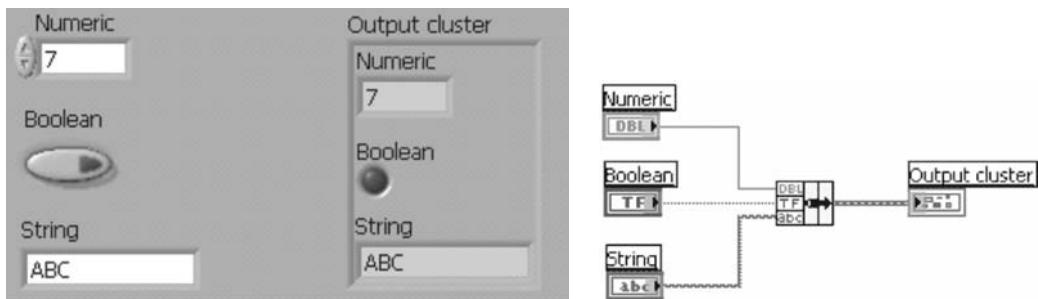
- Extract individual data elements from a cluster.
- Add individual data elements to a cluster.
- Break a cluster out into its individual data elements.

The *Bundle* function assembles individual components into a single new cluster and allows you to replace elements in an existing order. The *Unbundled* function splits a cluster into its individual components. When it is required to operate on a few elements and not the entire cluster elements, you use the *Bundle By Name* function. They are referenced by names rather than by position. The *Unbundle By Name* function returns the cluster elements whose names are specified.

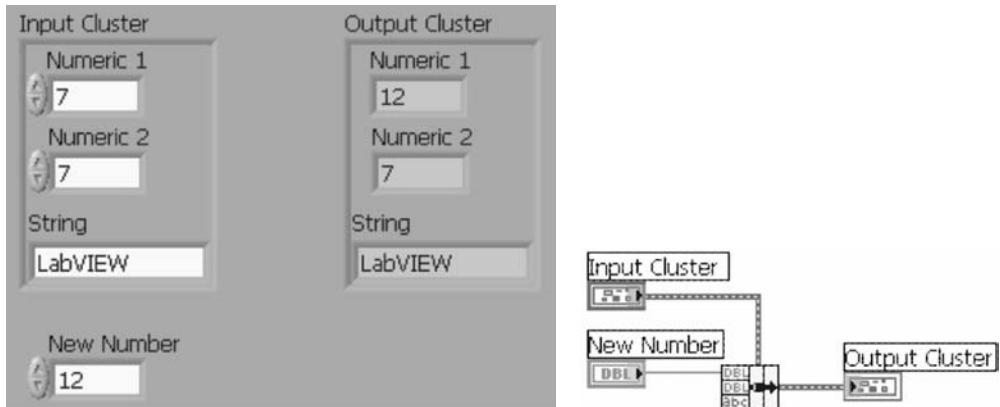
## 6.6 ASSEMBLING CLUSTERS

The *Bundle* function assembles a cluster from individual elements as shown in Figure 6.7. You also can use this function to change the values of individual elements in an existing cluster without

having to specify new values for all elements. To do so, wire the cluster you want to change to the middle cluster terminal of this function. When you wire a cluster to this function, the function resizes automatically to display inputs for each element in the cluster. The connector pane displays the default data types for this polymorphic function. In Figure 6.8 the input cluster consists of two numeric controls and a string control. The value of the first numeric control is modified without altering other values.



**Figure 6.7** Bundling of cluster from individual elements.



**Figure 6.8** Changing a value in the existing cluster.

Functions bundling elements into clusters lets you create clusters programmatically. Complete the following steps to bundle elements into a cluster.

**Step 1:** Place the *Bundle* function on the block diagram.

**Step 2:** If necessary, resize the *Bundle* function to include the number of inputs you intend to use as elements in the cluster. You cannot leave an input unwired.

**Step 3:** Wire front panel control terminals or outputs from VIs and functions to the *element* inputs of the *Bundle* function. The order in which you wire the inputs determines the cluster element order.

**Step 4:** Right-click the *Output Cluster* terminal and select *Create»Indicator*. LabVIEW returns the bundled cluster in the cluster output.

The *Bundle By Name* function is used to replace one or more elements in an existing cluster. This function refers to cluster elements by name instead of by their position in the cluster. After you wire the node to an input cluster, right-click the name terminals to select elements from the shortcut menu. You also can use the operating tool to click the name terminals and select from a list of cluster elements. All inputs are required. The connector pane displays the default data types for this polymorphic function. Figure 6.9 shows a *Bundle By Name* function used to modify the values of an existing input cluster which contains a string control and a numeric control. The new value for both the elements must be given, otherwise LabVIEW shows an error.

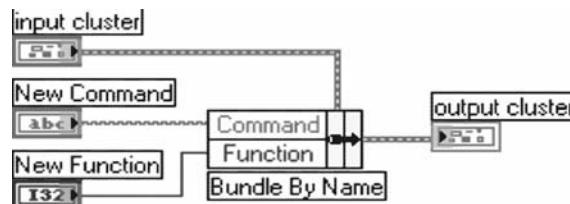


Figure 6.9 Modifying a cluster using *Bundle By Name* function.

## 6.7 DISASSEMBLING CLUSTERS

The *Unbundle* function splits a cluster into each of its individual elements. When you wire a cluster to this function, the function resizes automatically to display outputs for each element in the cluster you wired as shown in Figure 6.10. The connector pane displays the default data types for this polymorphic function. Unbundling elements from clusters accesses and arranges all elements in a cluster in their cluster element order. After you unbundle elements from clusters, you can wire each element to VIs, functions and indicators. This method of unbundling a cluster is useful if you need to access all the elements in a cluster. You also can unbundle all the elements from a cluster by name. The steps to unbundle elements from a cluster is place the *unbundled* function on the block diagram and then wire a cluster to the *unbundle* function. The data type representation of every element appears as element outputs.

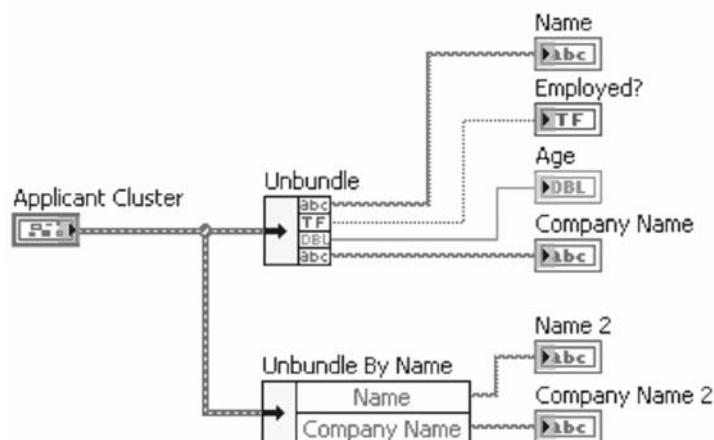


Figure 6.10 *Unbundle* function and *Unbundle By Name* function.

The *Unbundle By Name* function returns the cluster elements whose names you specify. You do not have to keep track of the order of the elements within the cluster. This function does not require the number of elements to match the number in the cluster. After you wire a cluster to this function, you can select an individual element from the function. The connector pane displays the default data types for this polymorphic function as shown in Figure 6.10.

Unbundling elements from clusters by name accesses and arranges the elements in a cluster by name in their cluster element order. A cluster element must have a label for you to unbundle the element by name. After you unbundle an element(s) from a cluster by name, you can wire the element(s) to a VI, function and indicator. This method of unbundling a cluster is useful if you need to access one element from a cluster that includes elements of the same data type. You also can unbundle all the elements from a cluster without using the name. The steps to unbundle elements from a cluster by name are first place the *Unbundle By Name* function on the block diagram. Then wire a cluster to the *Unbundle By Name* function. The first element in the cluster element order appears as an *element* output. Select an individual element from the function in the following ways:

1. Resize the function until the name of the element you want appears.
2. Right-click an *element* output, select *Select Item* from the shortcut menu, and select the element you want.
3. Use the operating tool to click an *element* output and select the element you want from the shortcut menu.

## 6.8 CONVERSION BETWEEN ARRAYS AND CLUSTERS

LabVIEW has many more functions for arrays than clusters and it is often required to change array to clusters and clusters to arrays. A cluster can be converted into an array first and converted back to a cluster after performing the required operation from the available array functions. You can convert a cluster with elements of the same data type to an array using the *Cluster To Array* function as shown in Figure 6.11 and use array functions to manipulate the contents. This function cannot be used on a cluster of arrays, since LabVIEW does not allow an array or arrays type of structure.

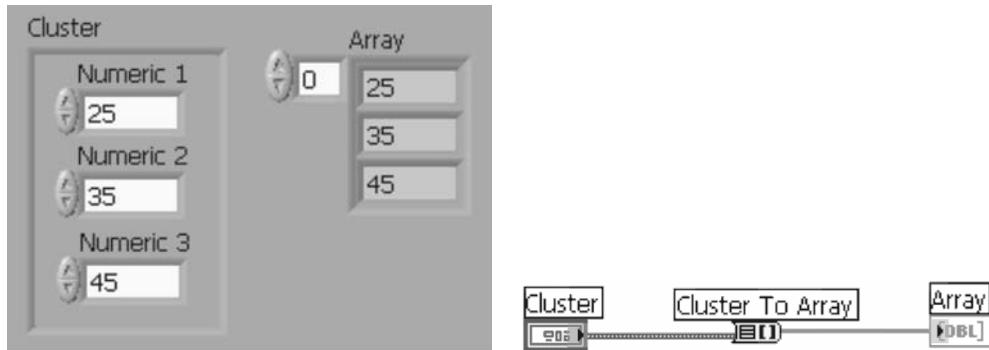


Figure 6.11 Converting a cluster into an array.

You can convert a cluster with elements of the same type to an array and use array functions to manipulate the contents. Complete the following steps to convert a cluster to an array.

**Step 1:** Place a cluster on the front panel.

**Step 2:** Place the *Cluster To Array* function on the block diagram.

**Step 3:** Wire the cluster to the *Cluster To Array* function.

**Step 4:** Right-click the *Cluster To Array* function and select *Create»Indicator* from the shortcut menu to create an array indicator.

**Step 5:** Run the VI from the front panel. The array indicator displays the values of the cluster.

You also can convert an array to a cluster using the *Array To Cluster* function. This function converts an  $n$  element, one-dimensional array into a cluster of  $n$  element of the same type. This function is useful when you would like to display the elements of the same type in a front panel but still want to manipulate the elements on the block diagram by their index values. Clusters do not size automatically, you need to specify the cluster size by popping up on function. The default cluster size is 9, and the maximum size permitted is 256.

The *Build Cluster Array* function is used to create an array of clusters where each cluster contains an array. This function does it by first bundling each component input into a cluster and then assembling all component clusters into an array of clusters.

The *Index and Build Cluster Array* function indexes a set of arrays and creates a cluster array in which the  $i$ th element contains the  $i$ th element of each input array. All array inputs need not be of the same type and the function yields a cluster array containing one element from each input array.

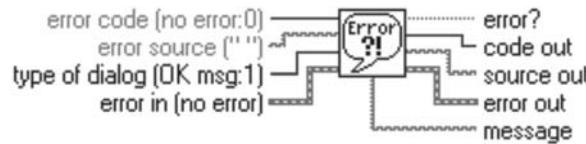
## 6.9 ERROR HANDLING

By default LabVIEW automatically handles any error that occurs when a VI runs by suspending execution, highlighting the subVI or function where the error occurred, and displaying a dialog box. You can choose other error handling methods. For example, if an I/O VI on the block diagram times out, you might not want the entire application to stop. You also might want the VI to retry for a certain period of time. In LabVIEW, you can make these error handling decisions on the block diagram of the VI.

VI's and functions return errors in one of two ways—with numeric error codes or with an error cluster. Typically, functions use numeric error codes, and VIs use an error cluster, usually with error inputs and outputs. Error handling in LabVIEW follows the data flow model. Just as data flows through a VI, so can error information. Wire the error information from the beginning of the VI to the end. Include an error handler VI at the end of the VI to determine if the VI ran without errors. Use the error in and error out clusters in each VI you use or build to pass error information through the VI.

As the VI runs, LabVIEW tests for errors at each execution node. If LabVIEW does not find any errors, the node executes normally. If LabVIEW detects an error, the node passes the error to the next node without executing. The next node does the same thing and so on. Use the simple

error handler VI, shown in Figure 6.12, to handle the error at the end of the execution flow. The simple error handler VI is located on the *Functions»All Functions»Time & Dialog* palette. Wire the error cluster to the error in input.



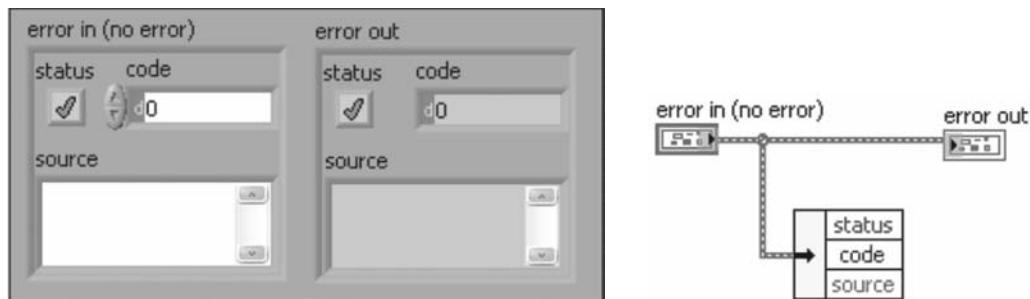
**Figure 6.12** Simple error handler.

## 6.10 ERROR CLUSTER

Error clusters tell you why and where errors occur. When you perform any kind of I/O, consider the possibility that errors will occur. Almost all I/O functions return error information. Include error checking in VIs, especially for I/O operations such as file, serial, instrumentation, data acquisition, and communication operations, and provide a mechanism to handle errors appropriately. No matter how confident you are in the VI you create, you cannot predict every problem a user might encounter. Without a mechanism to check for errors, you know only that the VI does not work properly. Checking for errors in VIs can help you identify the following problems:

- You initialized communications incorrectly or wrote improper data to an external device.
- An external device lost power, is broken, or is not working properly.
- You upgraded the operating system software which changed the path to a file or the functionality of a VI or library. You might notice a problem in a VI or a system program.

The error clusters located on the *Functions»All Functions»Array & Cluster* palette include the following components of information which are also shown in Figure 6.13.



**Figure 6.13** Error clusters in LabVIEW.

- **status** is a Boolean value that reports True if an error occurred. Most VIs, functions and structures that accept Boolean data also recognize this parameter. For example, you can wire an error cluster to the Boolean inputs of the *Stop*, *Quit LabVIEW*, or *Select* functions. If an error occurs, the error cluster passes a True value to the function.

- **code** is a 32-bit signed integer that identifies the error numerically. A non-zero error code coupled with a **status** of False signals a warning rather than a fatal error.
- **source** is a string that identifies where the error occurred. Use the error cluster controls and indicators to create error inputs and outputs in subVIs.

When an error occurs, right-click within the cluster border and select *Explain Error* from the shortcut menu to open the *Explain Error* dialog box. The *Explain Error* dialog box contains information about the error. The shortcut menu includes an *Explain Warning* option if the VI contains warnings but no errors. You also can access the *Explain Error* dialog box from the *Help»Explain Error* menu.

---

## SUMMARY

---

- Clusters group data elements of mixed types.
- Elements of clusters must be all controls or all indicators or constants.
- The size of components in a cluster is fixed.
- Cluster elements are accessed through the cluster order.
- If a front panel contains more than 28 controls and indicators that you want to use programmatically, group some of them into a cluster and assign the cluster to a terminal on the connector pane to eliminate clutter on the block diagram.
- To create a cluster control or indicator, select a cluster on the *Functions»All Functions»Array & Cluster* palette, place it on the front panel, and drag controls or indicators into the cluster shell.
- Use the cluster functions located on the *Functions»All Functions»Cluster* palette to create and manipulate clusters.
- Error checking tells you why and where errors occur.
- The error cluster reports the status, code and source of the error.
- Use the error cluster controls and indicators to create error inputs and outputs in subVIs.
- Arrays and clusters are inter-convertible but only under certain conditions.

---

## MISCELLANEOUS SOLVED PROBLEMS

---

**Problem 6.1** Create a VI to check whether the cluster elements are in range or not. Specify the upper and lower limits. Display the coerced output and a cluster of LEDs to indicate whether a particular cluster element is in the range or not.

**Solution** To solve the problem build the front panel and the block diagram as shown in Figures P6.1(a) and P6.1(b).

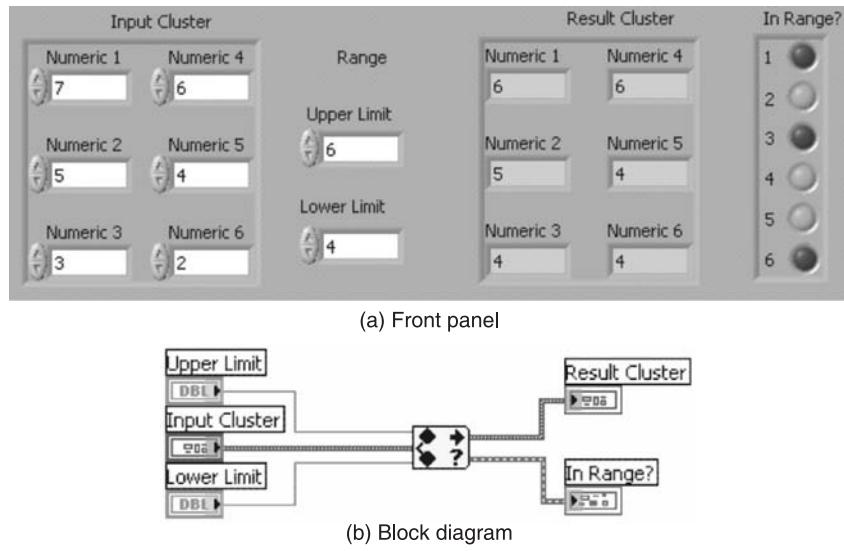


Figure P6.1

**Problem 6.2** Create a VI to compare clusters and Switch ON an LED in the output cluster if the  $n$ th element of cluster 1 is greater than the  $n$ th element of the cluster 2.

**Solution** The front panel and the block diagram to solve the problem are shown in Figure P6.2(a) and P6.2(b).

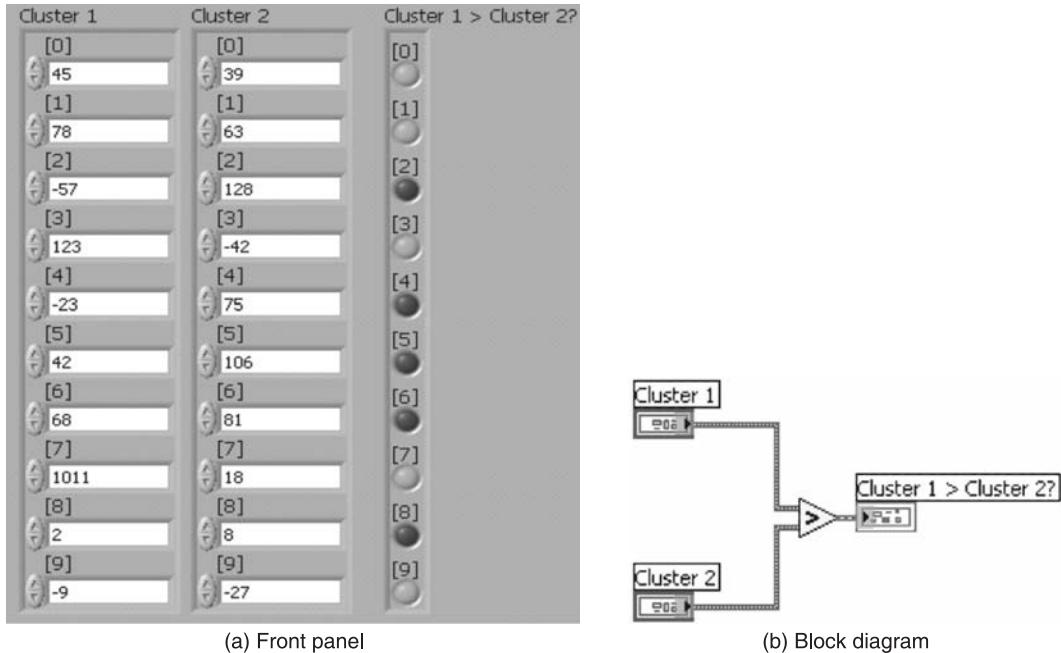


Figure P6.2

**Problem 6.3** Create a VI to add a value with every element of an available cluster. (Adding a numeric to a cluster results in the addition of the numeric to each element in the cluster.)

**Solution** To solve the problem build the front panel and the block diagram as shown in Figures P6.3(a) and P6.3(b).

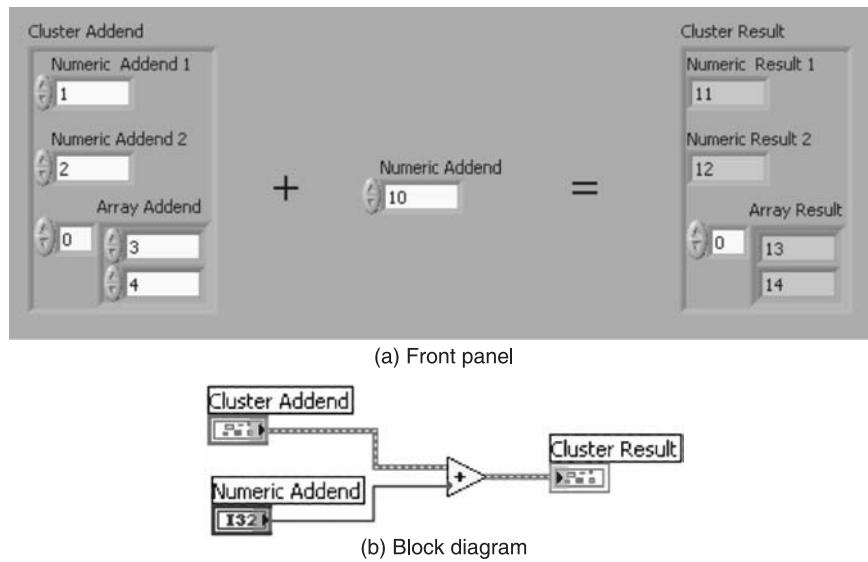


Figure P6.3

**Problem 6.4** Create a VI consisting of two clusters of LEDs. Perform the AND operation between the clusters and display the output in another cluster of LEDs. (When comparing clusters, the *And* function compares each element with its corresponding value in the second cluster.)

**Solution** Build the front panel and the block diagram as shown in Figures P6.4(a) and P6.4(b) to solve the problem.

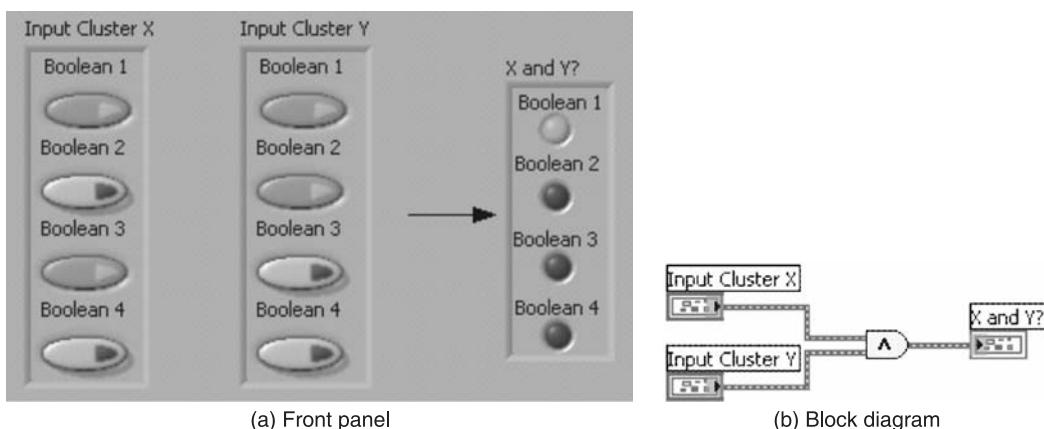


Figure P6.4

**Problem 6.5** Create a VI to compare the elements of two clusters. If the values of corresponding elements of both the VIs are the same, switch ON an LED in the output cluster.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P6.5(a) and P6.5(b).

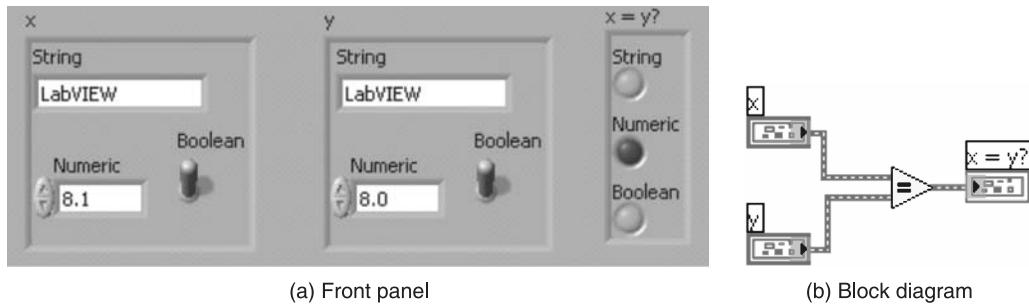


Figure P6.5

**Problem 6.6** Create a VI to select between two input clusters using a toggle switch and display in an output cluster.

**Solution** Build the front panel and the block diagram as shown in Figure P6.6(a) and P6.6(b) to solve the problem.

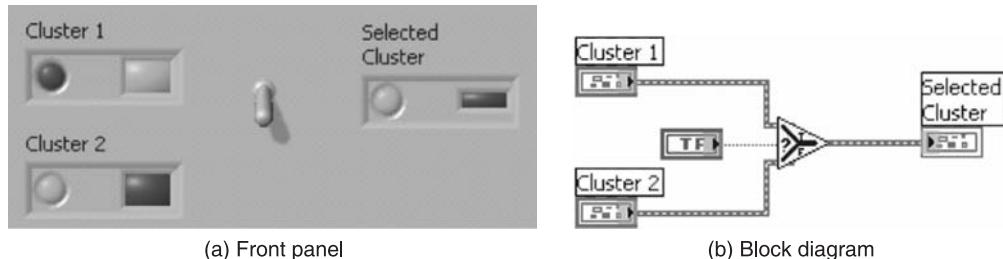
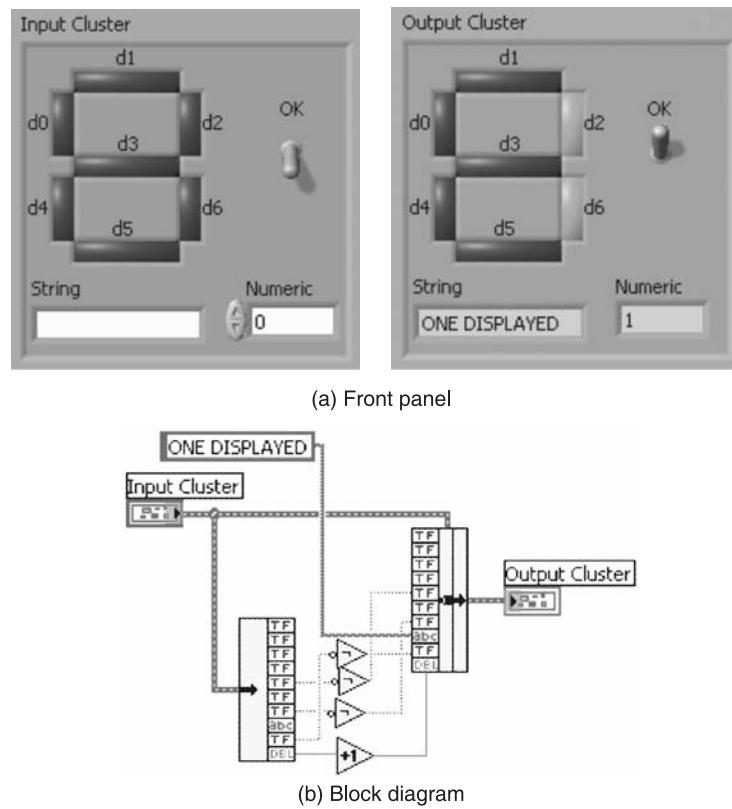


Figure P6.6

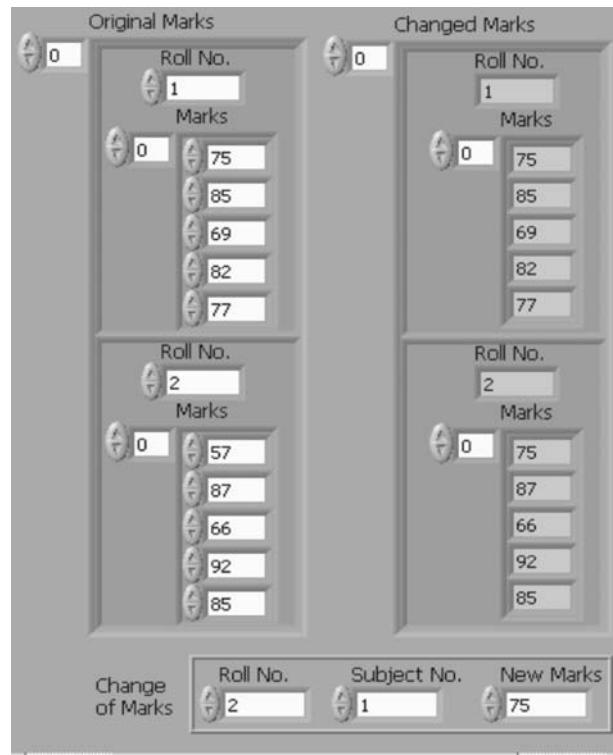
**Problem 6.7** Build a cluster control which consists of a seven-segment LED display, a switch, a string control and a numeric control. Split the cluster elements using the *Unbundle* function and alter the values of some of the cluster controls. Bundle them again and display in a cluster indicator.

**Solution** Create the front panel and the block diagram to solve the LED display problem as shown in Figures P6.7(a) and P6.7(b).

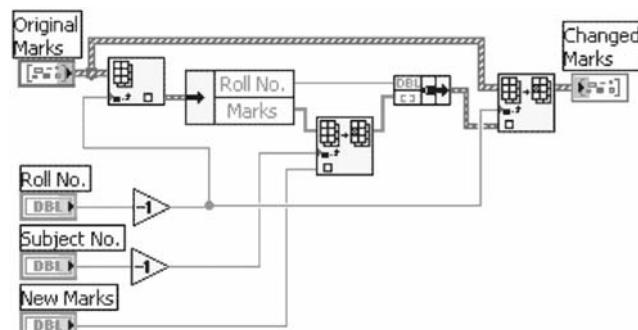
**Figure P6.7**

**Problem 6.8** Build an array of cluster controls in which each cluster consists of a numeric control and a 1D numeric array (with 5 elements). This forms a database of marks of students. The numeric control indicates the roll number and the array indicates the test marks of five subjects. Build logic to modify the mark in a particular subject of a particular student. Input the roll number, subject in which mark is to be changed and the new marks. Display the changed database on a separate array indicator.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P6.8(a) and P6.8(b).



(a) Front panel



(b) Block diagram

**Figure P6.8**

## REVIEW QUESTIONS

1. What is the function of a cluster?
2. Differentiate an array from a cluster.
3. Explain how clusters can reduce the number of terminals of a subV.I.
4. What is cluster order? Explain why it is important.

5. What is the difference between a *Bundle* and *Bundle By Name* functions?
6. Explain the method of changing the value of an element in an existing cluster.
7. Explain the use of error clusters with the help of an example.
8. With the help of an example explain assembling and disassembling clusters.

---

### EXERCISES

---

1. Build a cluster control which consists of a thermometer, a tank and a gauge. Build a cluster constant which consists of a scaling factor for all the three cluster controls. Display the scaled values in a cluster indicator.
2. Create a cluster which consists of a numeric control, two Boolean controls and a slide control. Modify the values of the cluster control and display them in a cluster indicator.
3. In Problem 6.8 change the 1D numeric array into 2D numeric array. The 2D array is to provide marks obtained in three tests.



# PLOTTING DATA

---

## 7.1 INTRODUCTION

Graphical display of data is an important aspect of programming in LabVIEW. A good knowledge of arrays and clusters is important for graphical operations. VIs with graph usually collects the data in an array and then plots the data to the graph to obtain a waveform. Two-dimensional X-Y displays are required in most situations. Charts and graphs let you display plots of data in a graphical form. Charts interactively plot data, appending new data to old so that you can see the current value in the context of previous data, as the new data become available. Graphs plot pre-generated arrays of values in a more traditional fashion without retaining previously-generated data.

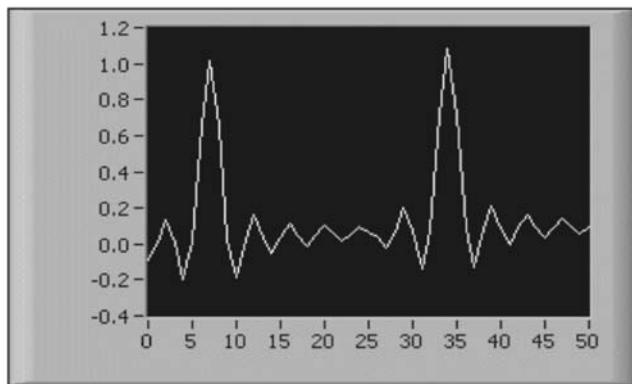
## 7.2 TYPES OF WAVEFORMS

LabVIEW includes the following types of graphs and charts:

- **Waveform graphs and charts:** Display data typically acquired at a constant rate.
- **XY Graphs:** Display data acquired at a non-constant rate and data for multivalued functions.
- **Intensity graphs and charts:** Display 3D data on a 2D plot by using color to display the values of the third dimension.
- **Digital waveform graphs:** Display data as pulses or groups of digital lines.
- **Windows 3D Graphs:** Display 3D data on a 3D plot in an ActiveX object on the front panel.

### 7.3 WAVEFORM GRAPHS

LabVIEW includes the waveform graph and chart to display data typically acquired at a constant rate. The waveform graph displays one or more plots of evenly sampled measurements. The waveform graph plots only single-valued functions, as in  $y = f(x)$ , with points evenly distributed along the  $x$ -axis, such as acquired time-varying waveforms. Figure 7.1 shows an example of a waveform graph. The waveform graph can display plots containing any number of points. The graph also accepts several data types, which minimizes the extent to which you must manipulate data before you display it.



**Figure 7.1** Waveform Graph.

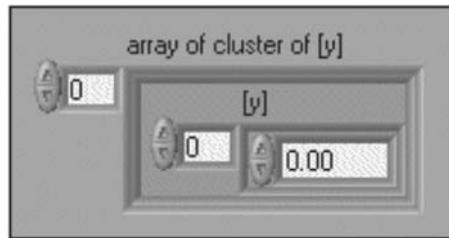
#### 7.3.1 Displaying a Single Plot on Waveform Graphs

The waveform graph accepts several data types for single-plot waveform graphs. The graph accepts a single array of values, interprets the data as points on the graph and increments the  $x$  index by one starting at  $x = 0$ . The graph accepts a cluster of an initial  $x$  value, a delta  $x$  and an array of  $y$  data. The graph also accepts the waveform data type, which carries the data, start time and delta  $t$  of a waveform. The waveform graph also accepts the dynamic data type, which is for use with Express VIs. In addition to the data associated with a signal, the dynamic data type includes attributes that provide information about the signal such as the name of the signal or the date and time the data was acquired. Attributes specify how the signal appears on the waveform graph. When the dynamic data type includes a single numeric value, the graph plots the single value and automatically formats the plot legend and  $x$ -scale time stamp. When the dynamic data type includes a single channel, the graph plots the whole waveform and automatically formats the plot legend and  $x$ -scale time stamp.

#### 7.3.2 Displaying Multiple Plots on Waveform Graphs

The waveform graph accepts several data types for displaying multiple plots. The waveform graph accepts a 2D array of values, where each row of the array is a single plot. The graph interprets the data as points on the graph and increments the  $x$  index by one, starting at  $x = 0$ . Wire a 2D array data type to the graph, right-click the graph and select *Transpose Array* from the shortcut menu to handle each column of the array as a plot. This is particularly useful when you sample multiple

channels from a DAQ device because the device can return the data as 2D arrays with each channel stored as a separate column. The waveform graph also accepts a cluster of an initial  $x$  value, a delta  $x$  value, and a 2D array of  $y$  data. The graph interprets the  $y$  data as points on the graph and increments the  $x$  index by delta  $x$ , starting at the initial  $x$  value. This data type is useful for displaying multiple signals that are sampled at the same regular rate. The waveform graph accepts a plot array where the array contains clusters. Each cluster contains a 1D array that contains the  $y$  data. The inner array describes the points in a plot, and the outer array has one cluster for each plot. The front panel in Figure 7.2 shows the array of the  $y$  cluster.



**Figure 7.2** Array of the  $y$  cluster.

Use a plot array instead of a 2D array if the number of elements in each plot is different. For example, when you sample data from several channels using different time amounts from each channel, use this data structure instead of a 2D array because each row of a 2D array must have the same number of elements. The number of elements in the interior arrays of an array of clusters can vary. The waveform graph accepts a cluster of an initial  $x$  value, a delta  $x$  value, and array that contains clusters. Each cluster contains a 1D array that contains the  $y$  data. You use the *Bundle* function to bundle the arrays into clusters and you use the *Build Array* function to build the resulting clusters into an array. You also can use the *Build Cluster Array* function which creates arrays of clusters that contain the inputs you specify.

The waveform graph accepts an array of clusters of an  $x$  value, a delta  $x$  value and an array of  $y$  data. This is the most general of the multiple-plot waveform graph data types because you can indicate a unique starting point and increment for the  $x$ -scale of each plot. The waveform graph also accepts the dynamic data type, which is for use with Express VIs. In addition to the data associated with a signal, the dynamic data type includes attributes that provide information about the signal, such as the name of the signal or the date and time the data was acquired. Attributes specify how the signal appears on the waveform graph. When the dynamic data type includes multiple channels, the graph displays a plot for each channel and automatically formats the plot legend and  $x$ -scale time stamp.

## 7.4 WAVEFORM CHARTS

The waveform chart is a special type of numeric indicator that displays one or more plots of data typically acquired at a constant rate.

Waveform charts can display single or multiple plots. Figure 7.3 shows the elements of a multiplot waveform chart. Two plots are displayed: *Raw Data* and *Running Avg*. The waveform chart maintains a history of data or buffer from previous updates.

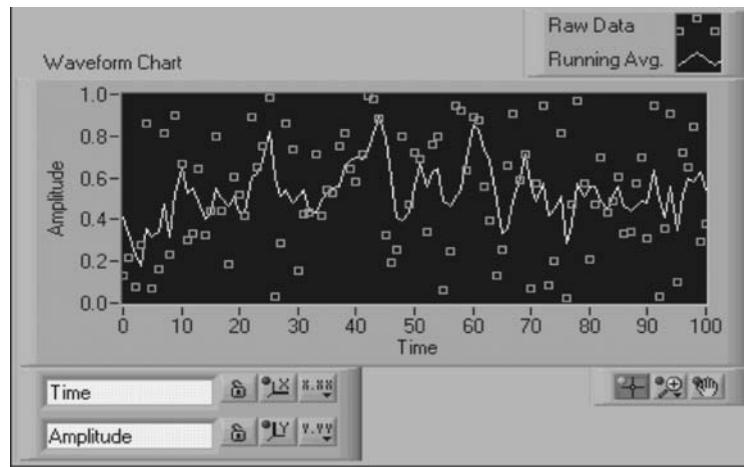


Figure 7.3 A waveform chart.

#### 7.4.1 Displaying a Single Plot on Waveform Charts

You can wire a scalar output directly to a waveform chart. The waveform chart terminal shown in Figure 7.4 matches the input data type. If you pass the chart a single value or multiple values at a time, LabVIEW interprets the data as points on the chart and increments the  $x$  index by one starting at  $x = 0$ . The chart treats these inputs as new data for a single plot. The waveform chart accepts the waveform data type which carries the data, start time and delta  $t$  of a waveform. Use the *Build Waveform* function to plot time on the  $x$ -axis of the chart and automatically use the correct interval between markers on the  $x$ -scale of the chart. A waveform that specifies  $t_0$  and a single-element  $Y$  array is useful for plotting data that is not evenly sampled because each data point has its own time stamp.

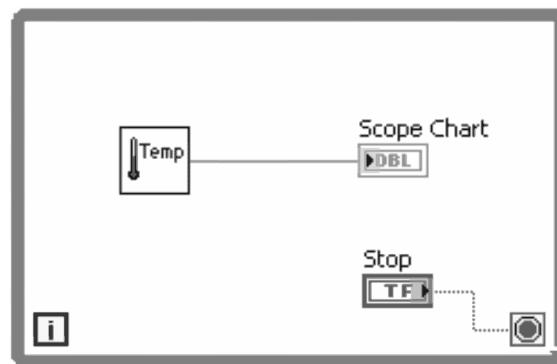
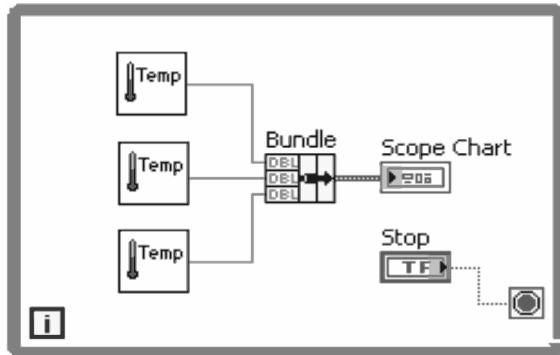


Figure 7.4 Scalar output wired directly to a waveform chart.

#### 7.4.2 Displaying Multiple Plots on Waveform Charts

Waveform charts can display multiple plots together using the *Bundle* function located on the *Cluster* palette. In Figure 7.5, the *Bundle* function bundles the outputs of the three VIs to plot on

the waveform chart. The waveform chart terminal changes to match the output of the *Bundle* function. To add more plots, use the positioning tool to resize the *Bundle* function.



**Figure 7.5** Bundle function bundles the outputs to plot on the waveform chart.

To pass data for multiple plots to a waveform chart, you can bundle the data together into a cluster of scalar numeric values, where each numeric represents a single point for each of the plots. If you want to pass multiple points per plot in a single update, wire an array of clusters of numeric values to the chart. Each numeric represents a single *y* value point for each of the plots. You can use the waveform data type to create multiple plots on a waveform chart. Use the *Build Waveform* function to plot time on the *x*-axis of the chart and automatically use the correct interval between markers on the *x*-scale of the chart. A 1D array of waveforms that each specifies  $t_0$  and a single-element *Y* array is useful for plotting data that is not evenly sampled because each data point has its own time stamp.

If you cannot determine the number of plots you want to display until run time, or you want to pass multiple points for multiple plots in a single update, wire a 2D array of numeric values or waveforms to the chart. By default, the waveform chart treats each column in the array as a single plot. Wire a 2D array data type to the chart, right-click the chart, and select *Transpose Array* from the shortcut menu to treat each row in the array as a single plot.

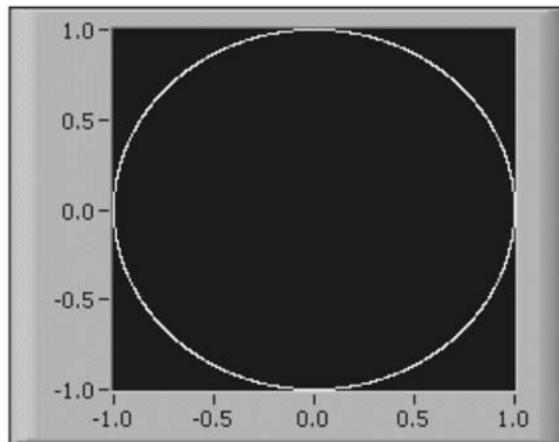
## 7.5 WAVEFORM DATA TYPE

The waveform data type carries the data, start time and delta *t* of a waveform. You can create a waveform using the *Build Waveform* function. Many of the VIs and functions you use to acquire or analyze waveforms accept and return waveform data by default. When you wire waveform data to a waveform graph or chart, the graph or chart automatically plots a waveform based on the data, start time and delta *x* of the waveform. When you wire an array of waveform data to a waveform graph or chart, the graph or chart automatically plots all waveforms.

## 7.6 XY GRAPHS

The *XY* graph is a general-purpose, Cartesian graphing object that plots multivalued functions, such as circular shapes or waveforms with a varying time base. It displays any set of points, evenly

sampled or not. You also can display Nyquist planes, Nichols planes, S planes and Z planes on the XY graph. Lines and labels on these planes are the same color as the Cartesian lines, and you cannot modify the plane label font. Figure 7.6 shows an example of an XY graph. The XY graph can display plots containing any number of points. It also accepts several data types, which minimizes the extent to which you must manipulate data before you display it.



**Figure 7.6** An XY graph.

#### 7.6.1 Displaying a Single Plot on XY Graphs

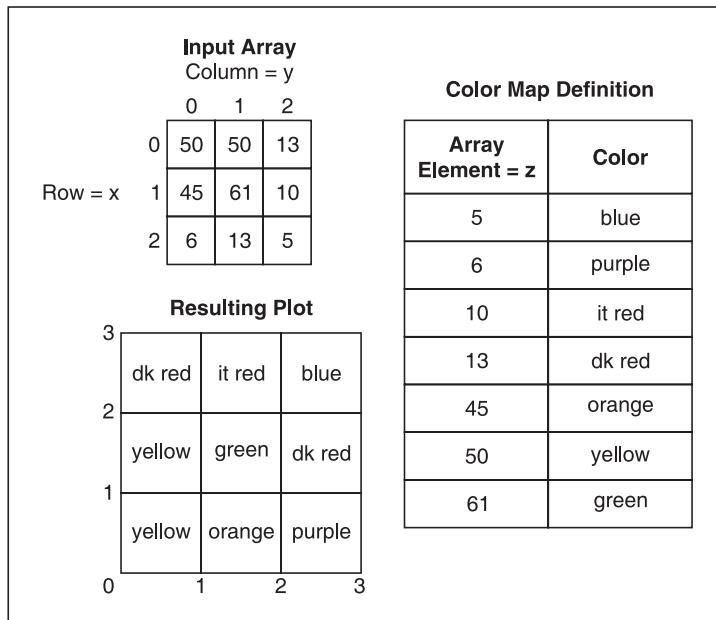
The XY graph accepts three data types for single-plot XY graphs. The XY graph accepts a cluster that contains an x array and a y array. The XY graph also accepts an array of points where a point is a cluster that contains an x value and a y value. It also accepts an array of complex data in which the real part is plotted on the x-axis and the imaginary part is plotted on the y-axis.

#### 7.6.2 Displaying Multiple Plots on XY Graphs

The XY graph accepts three data types for displaying multiple plots. It accepts an array of plots where a plot is a cluster that contains an x array and a y array. The XY graph also accepts an array of clusters of plots where a plot is an array of points. A point is a cluster that contains an x value and a y value. The XY graph also accepts an array of clusters of plots where a plot is an array of complex data, in which the real part is plotted on the x-axis and the imaginary part is plotted on the y-axis.

### 7.7 INTENSITY GRAPHS AND CHARTS

Use the intensity graph and chart to display 3D data on a 2D plot by placing blocks of color on a Cartesian plane. For example, you can use an intensity graph or chart to display patterned data, such as temperature patterns and terrain, where the magnitude represents altitude. The intensity graph and chart accept a 3D array of numbers. Each number in the array represents a specific color. The indexes of the elements in the 2D array set the plot locations for the colors. Figure 7.7 shows the concept of the intensity chart operation.



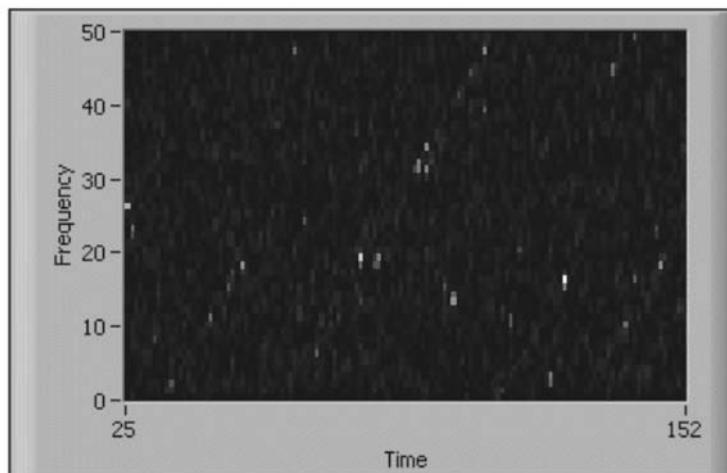
**Figure 7.7** Intensity chart operation.

The rows of the data pass into the display as new columns on the graph or chart. If you want rows to appear as rows on the display, wire a 2D array data type to the graph or chart, right-click the graph or chart, and select *Transpose Array* from the shortcut menu. The array indexes correspond to the lower-left vertex of the block of color. The block of color has a unit area which is the area between the two points, as defined by the array indexes. The intensity graph or chart can display up to 256 discrete colors.

### 7.7.1 Intensity Charts

After you plot a block of data on an intensity chart, the origin of the Cartesian plane shifts to the right of the last data block. When the chart processes new data, the new data values appear to the right of the old data values. When a chart display is full, the oldest data values scroll off the left side of the chart. This behavior is similar to the behavior of a strip chart. Figure 7.8 shows an example of an intensity chart.

The intensity chart shares many of the optional parts of the waveform chart, including the scale legend and graph palette, which you can show or hide by right-clicking the chart and selecting *Visible Items* from the shortcut menu. In addition, because the intensity chart includes color as a third dimension, a scale similar to a color ramp control defines the range and mappings of values to colors. Like the waveform chart, the intensity chart maintains a history of data, or buffer, from previous updates. Right-click the chart and select *Chart History Length* from the shortcut menu to configure the buffer. The default size for an intensity chart is 128 data points. The intensity chart display can be memory intensive.



**Figure 7.8** An intensity chart.

### 7.7.2 Intensity Graphs

The intensity graph works the same as the intensity chart, except it does not retain previous data values and does not include update modes. Each time new data values pass to an intensity graph, the new data values replace old data values. Like other graphs, the intensity graph can have cursors. Each cursor displays the  $x$ ,  $y$  and  $z$  values for a specified point on the graph.

### 7.7.3 Using Color Mapping with Intensity Graphs and Charts

An intensity graph or chart uses color to display 3D data on a 2D plot. When you set the color mapping for an intensity graph or chart, you configure the color scale of the graph or chart. The color scale consists of at least two arbitrary markers, each with a numeric value and a corresponding display color. The colors displayed on an intensity graph or chart correspond to the numeric values associated with the specified colors. Color mapping is useful for visually indicating data ranges, such as when plot data exceeds a threshold value. You can set the color mapping interactively for the intensity graph and chart the same way you define the colors for a color ramp numeric control.

You can set the color mapping for the intensity graph and chart programmatically by using the property node in two ways. Typically, you specify the value-to-color mappings in the property node. For this method, specify the Z scale: *Marker Values* property. This property consists of an array of clusters in which each cluster contains a numeric limit value and the corresponding color to display for that value. When you specify the color mapping in this manner, you can specify an upper out-of-range color using the Z scale: *High Color* property and a lower out-of-range color using the Z scale: *Low Color* property. The intensity graph and chart are limited to a total of 254 colors, with the lower and upper out-of-range colors bringing the total to 256 colors. If you specify more than 254 colors, the intensity graph or chart creates the 254-color table by interpolating among the specified colors.

If you display a bitmap on the intensity graph, you specify a color table using the *Color Table* property. With this method, you can specify an array of up to 256 colors. Data passed to the chart are mapped to indexes in this color table based on the color scale of the intensity chart. If the color scale ranges from 0 to 100, a value of 0 in the data is mapped to index 1, and a value of 100 is mapped to index 254 with interior values interpolated between 1 and 254. Anything below 0 is mapped to the out-of-range below color (index 0), and anything above 100 is mapped to the out-of-range above color (index 255).

## 7.8 DIGITAL WAVEFORM GRAPHS

Use the digital waveform graph to display digital data, especially when you work with timing diagrams or logic analyzers. The digital waveform graph accepts the digital waveform data type, the digital data type, and an array of those data types as an input. By default, the digital waveform graph collapses digital buses, so the graph plots digital data on a single plot. If you wire an array of digital data, the digital waveform graph plots each element of the array as a different plot in the order of the array.

The digital waveform graph in Figure 7.9 shows the front panel plots digital data on a single plot. The VI converts the numbers in the *Numbers* array to digital data and displays the binary representations of the numbers in the *Binary Representations* digital data indicator. In the digital graph, the number 0 appears without a top line to symbolize that all the bit values are zero. The number 255 appears without a bottom line to symbolize that all the bit values are 1. Right-click the y-scale and select *Expand Digital Buses* from the shortcut menu to plot each sample of digital data. Each plot represents a different bit in the digital pattern. The digital waveform graph in Figure 7.10 shows the front panel displays the six numbers in the *Numbers* array.

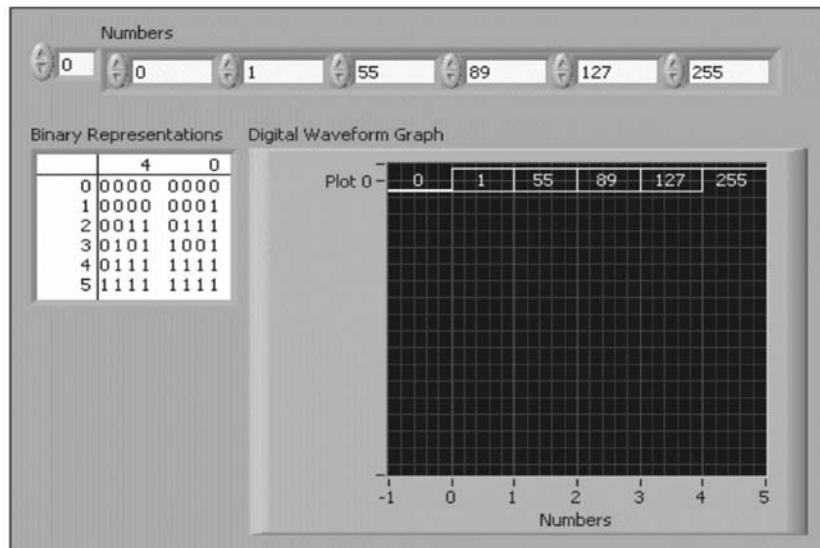
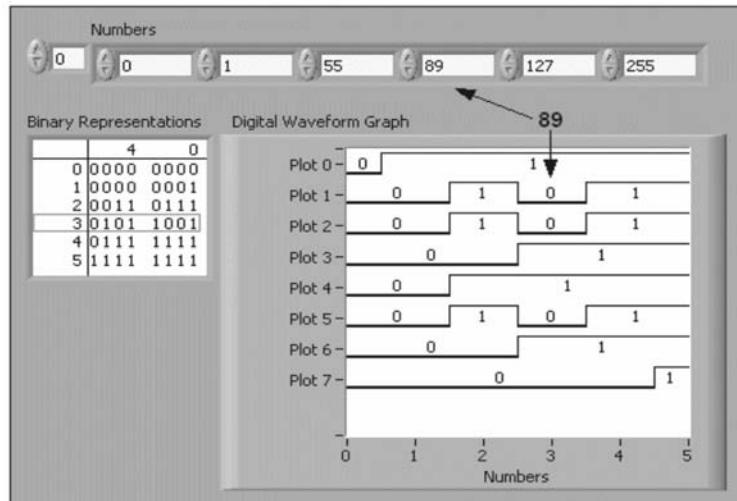
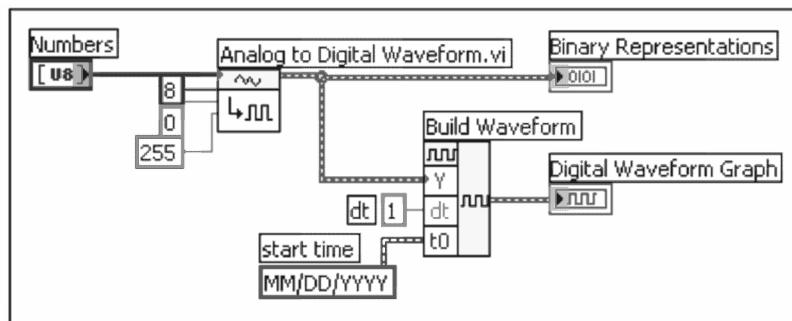


Figure 7.9 Digital waveform graph.



**Figure 7.10** Digital waveform graph with six numbers in the *Numbers* array.

The *Binary Representations* digital indicator displays the binary representations of the numbers. Each column in the table represents a bit. For example, the number 89 requires 7 bits of memory (the 0 in column 7 indicates an unused bit). Point 3 on the digital waveform graph plots the 7 bits necessary to represent the number 89 and a value of 0 to represent the unused eighth bit on plot 7. The VI in Figure 7.11 converts an array of numbers to digital data and uses the *Build Waveform* function to assemble the start time, delta  $t$ , and the numbers entered in a digital data control and to display the digital data.



**Figure 7.11** Build Waveform function.

### 7.8.1 Digital Waveform Data Type

The digital waveform data type carries start time, delta  $x$ , the data and the attributes of a digital waveform. You can use the *Build Waveform* function to create a digital waveform. When you wire digital waveform data to the digital waveform graph, the graph automatically plots a waveform based on the timing information and data of the digital waveform. Wire digital waveform data to a digital data indicator to view the samples and signals of a digital waveform.

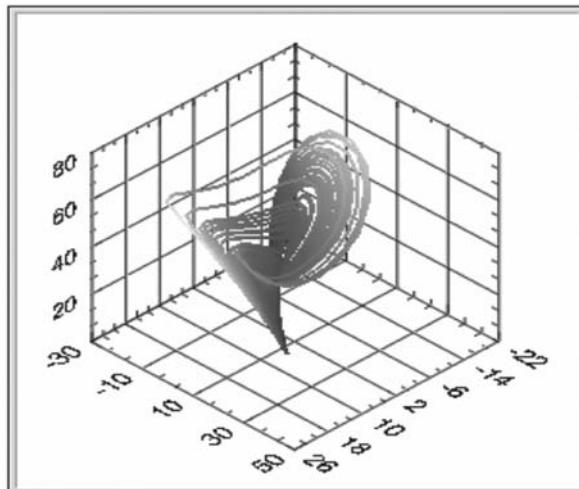
## 7.9 3D GRAPHS

For many real-world data sets such as the temperature distribution on a surface, joint time-frequency analysis and the motion of an airplane, you need to visualize data in three dimensions. With the 3D graphs, you can visualize three-dimensional data and alter the way that data appears by modifying the 3D graph properties.

LabVIEW includes the following types of 3D graphs:

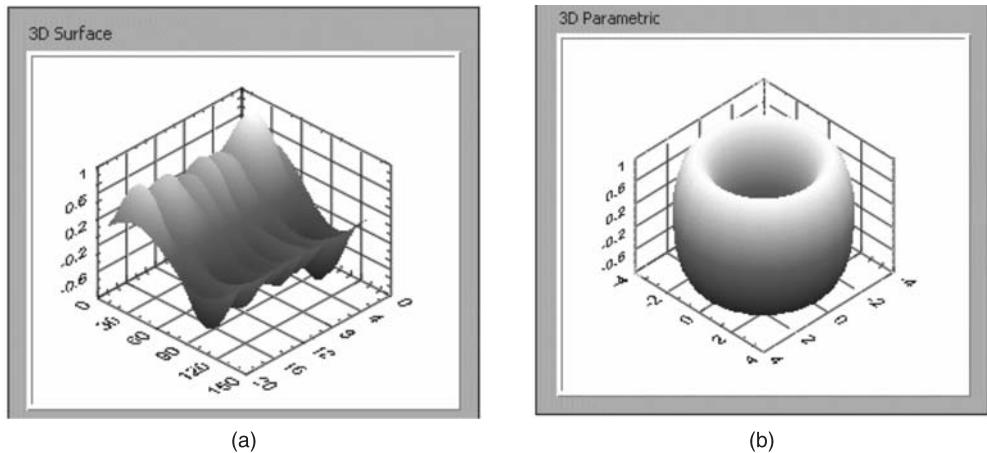
- **3D surface graph:** Draws a surface in 3D space.
- **3D parametric surface graph:** Draws a parametric surface in 3D space.
- **3D curve graph:** Draws a line in 3D space.

Use the 3D graphs in conjunction with the 3D Graph VIs to plot curves and surfaces. A curve contains individual points on the graph, each point having an  $x$ ,  $y$  and  $z$  coordinates. The VI then connects these points with a line. A curve is ideal for visualizing the path of a moving object, such as the flight path of an airplane. Figure 7.12 shows an example of a 3D curve graph.



**Figure 7.12** A 3D curve graph.

A surface plot uses  $x$ ,  $y$  and  $z$  data to plot points on the graph. The surface plot then connects these points forming a three-dimensional surface view of the data. For example, you could use a surface plot for terrain mapping. Figure 7.13(a) shows examples of a 3D surface graph and Figure 7.13(b) shows a 3D parametric surface graph. The 3D graphs use ActiveX technology and VIs that handle 3D representation. When you select a 3D graph, LabVIEW places an ActiveX container on the front panel that contains a 3D graph control. LabVIEW also places a reference to the 3D graph control on the block diagram. LabVIEW wires this reference to one of the three 3D Graph VIs.



**Figure 7.13** (a) A 3D surface graph and (b) A 3D parametric surface graph.

## 7.10 CUSTOMIZING GRAPHS AND CHARTS

Each graph and chart includes many options that you can use to customize appearance, convey more information, or highlight data. Although graphs and charts plot data differently, they have several common options that you access from the shortcut menu. However, some options are available only for a specific type of graph or chart.

### 7.10.1 Using Multiple X- and Y-Scales

All graphs support multiple *x*- and *y*-scales, and all charts support multiple *y*-scales. Use multiple scales on a graph or chart to display multiple plots that do not share a common *x*- or *y*-scale. Right-click the scale of the graph or chart and select *Duplicate Scale* from the shortcut menu to add multiple scales to the graph or chart.

### 7.10.2 Autoscaling

All graphs and charts can automatically adjust their horizontal and vertical scales to fit the data you wire to them. This behavior is called *autoscaling*. Right-click the graph or chart and select *X Scale»AutoScale X* or *Y Scale»AutoScale Y* from the shortcut menu to turn autoscaling ON or OFF. By default, autoscaling is enabled for the graph or chart. However, autoscaling can slow performance. Use the operating tool or the labeling tool to change the horizontal or vertical scale directly.

### 7.10.3 Formatting X- and Y-Scales

Use the *Format and Precision* page of the *Properties* dialog box to specify how the scales of the *x*-axis and *y*-axis appear on the graph or chart. By default, the *x*-scale is configured to use floating-point notation and have a label of time, and the *y*-scale is configured to use automatic formatting and have a label of amplitude. To configure the scales for the graph or chart, right-click the graph or chart and select *Properties* from the shortcut menu to display the *Graph Properties* dialog box or *Chart Properties* dialog box.

Use the *Format and Precision* page of the *Properties* dialog box to specify a numeric format for the scales of a graph or chart. Click the *Scales* tab to rename the scale and to format the appearance of the axis scale. By default, a graph or chart scale displays up to six digits before automatically switching to exponential notation. On the *Format and Precision* page, select *Advanced editing mode* to display the text options that let you enter format strings directly. Enter format strings to customize the appearance and numeric precision of the scales.

#### 7.10.4 Using the Graph Palette

Use the graph palette, shown as follows, to interact with a graph or chart while the VI is running.



With the graph palette, you can move cursors, zoom and pan the display. Right-click the graph or chart and select *Visible Items»Graph Palette* from the shortcut menu to display the graph palette. The graph palette appears with the following buttons, in order from left to right:

**Cursor movement tool** (graph only)—Moves the cursor on the display.

**Zoom**—Zooms in and out of the display.

**Panning tool**—Picks up the plot and moves it around on the display.

Click a button in the graph palette to enable moving the cursor, zooming the display, or panning the display. Each button displays a green LED when it is enabled.

#### 7.10.5 Customizing Graph and Chart Appearance

Customize the appearance of a graph or chart by showing or hiding options. Right-click the graph or chart and select *Visible Items* from the shortcut menu to display or hide the following options:

- **Plot legend**—Defines the color and style of plots. Resize the legend to display multiple plots.
- **Scale legend**—Defines labels for scales and configures scale properties.
- **Graph palette**—Allows you to move the cursor and zoom and pan the graph or chart while a VI runs.
- **X scale and Y scale**—Formats the *x*- and *y*-scales.
- **Cursor legend** (graph only)—Displays a marker at a defined point coordinate. You can display multiple cursors on a graph.
- **X scrollbar**—Scrolls through the data in the graph or chart. Use the scroll bar to view data that the graph or chart does not currently display.
- **Digital display** (waveform chart only)—Displays the numeric value of the chart.

#### 7.10.6 Exporting Images of Graphs, Charts, and Tables

You can include black and white images of graphs, charts, tables, and digital data and digital waveform controls and indicators into presentations, email, text documents, and so on. When you export a simplified image, LabVIEW exports only the control or indicator, digital display, plot

legend, and index display and does not export scrollbars, the scale legend, the graph palette or the cursor palette.

You can export images into the following formats:

Windows .emf, .bmp and .eps files

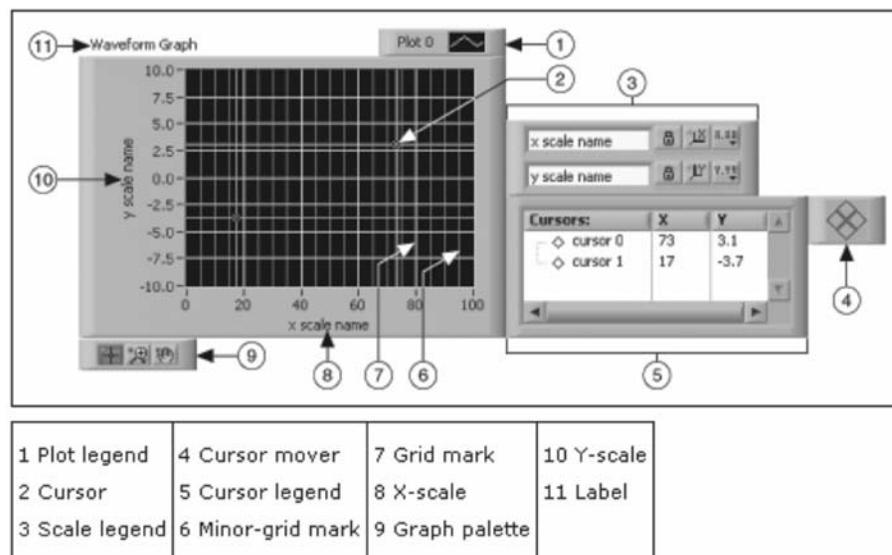
Mac .pict, .bmp and .eps files

Linux .bmp and .eps files

You can save the image to the clipboard or to disk.

## 7.11 CUSTOMIZING GRAPHS

Each graph includes options that you can use to customize the graph to match your data display requirements. For example, you can modify the behavior and appearance of graph cursors or configure graph scales. Figure 7.14 shows the elements of a graph.

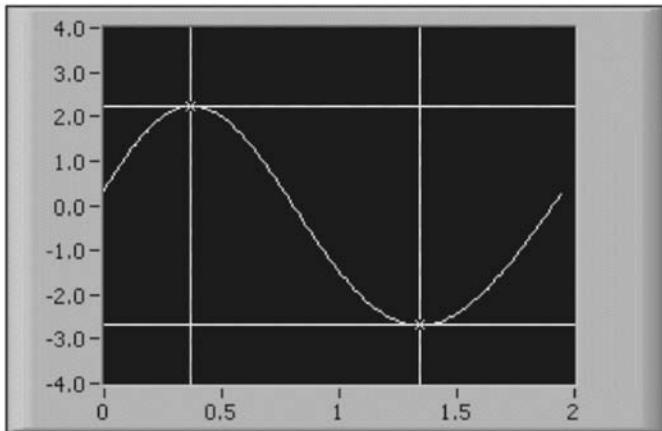


**Figure 7.14** The elements of a graph.

You add most of the items listed in the legend above by right-clicking the graph, selecting *Visible Items* from the shortcut menu, and selecting the appropriate element. Right-click the graph and select the option from the shortcut menu to set the graph option.

### 7.11.1 Using Graph Cursors

Use a cursor on a graph to read the exact value of a point on a plot or a point in the plot area. The cursor value displays in the cursor legend. Figure 7.15 shows an example of a graph using multiple cursors. Right-click the graph and select *Visible Items»Cursor Legend* from the shortcut menu to view the cursor legend. Add a cursor to the graph by right-clicking anywhere in the cursor legend, selecting *Create Cursor*, and selecting a cursor mode from the shortcut menu.



**Figure 7.15** Graph using multiple cursors.

The cursor position is defined by the cursor mode. The cursor includes the following modes:

- **Free**—Moves the cursor freely within the plot area, regardless of plot positions.
- **Single-Plot**—Positions the cursor only on the plot that is associated with the cursor. You can move the cursor along the associated plot. Right-click the cursor legend row and select **Snap To** from the shortcut menu to associate one or all plots with the cursor.
- **Multi-Plot**—Positions the cursor only on a specific data point in the plot area. The multi-plot cursor reports values at the specified  $x$ -value for all of the plots with which the cursor is associated. You can position the cursor on any plot in the plot area. Right-click the cursor legend row and select **Snap To** from the shortcut menu to associate one or all plots with the cursor. This mode is valid only for mixed signal graphs.

You can customize the appearance of the cursor in several ways. You can label the cursor on the plot, specify the color of the cursor, and specify line, point, and cursor style. Right-click the cursor legend row and select items from the shortcut menu to customize the cursor.

### 7.11.2 Using Graph Annotations

Use annotations on a graph to highlight data points in the plot area. The annotation includes a label and an arrow that identifies the annotation and data point. A graph can have any number of annotations. Right-click the graph and select *Data Operations»Create Annotation* from the shortcut menu to display the *Create Annotation* dialog box. Use the *Create Annotation* dialog box to specify the annotation name and how the annotation snaps to plots in the plot area. Use the *Lock Style* pull-down menu in the *Create Annotation* dialog box to specify how the annotation snaps to plots in the plot area. The *Lock Style* component includes the following options:

**Free**—Allows you to move the annotation anywhere in the plot area. LabVIEW does not snap the annotation to any plots in the plot area.

**Snap to All Plots**—Allows you to move the annotation to the nearest data point along any plot in the plot area.

**Snap to One Plot**—Allows you to move the annotation only along the specified plot.

Figure 7.16 shows an example of a graph using annotations. You can customize the behavior and appearance of the annotation in several ways. You can hide or show the annotation name or arrow in the plot area, specify the color of the annotation, and specify line, point, and annotation style. Right-click the annotation and select options from the shortcut menu to customize the annotation. To delete the annotation, right-click the annotation and select *Delete Annotation* from the shortcut menu. Right-click the graph and select *Data Operations»Delete All Annotations* from the shortcut menu to delete all annotations in the plot area.

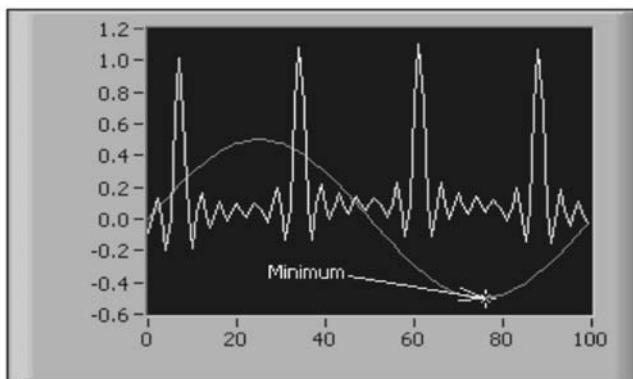


Figure 7.16 Graph with annotations.

## 7.12 CUSTOMIZING 3D GRAPHS

The 3D graphs have many options that you can use to customize them, including 3D plot styles, scale formatting, grids and plot projection. Because the 3D graphs use ActiveX technology and VIs that handle 3D representation, you set options for the 3D graphs differently than you set options for other graphs. While creating an application, use the ActiveX Property Brower to set properties for a 3D graph. Right-click the 3D graph and select *Property Brower* from the shortcut menu to display the ActiveX Property Brower. If you want to allow users to change common properties at run time or you need to set a property programmatically, use the 3D Graph Properties VIs.

## 7.13 CUSTOMIZING CHARTS

Unlike the graph, which displays new data that overwrites any stored data, the chart updates periodically and maintains a history of the data previously stored. You can customize the chart to match your data display requirements. Options available for all charts include a scroll bar, the scale legend, the graph palette, a digital display, and representation of scales with respect to time. You also can modify the behavior of chart history length, update modes and plot displays.

### 7.13.1 Configuring Chart History Length

LabVIEW stores data points already added to the chart in a buffer, or the chart history. The default size for a chart history buffer is 1024 data points. Right-click the chart and select *Chart History*

*Length* from the shortcut menu to configure the history buffer. You can view previously collected data using the chart scroll bar. Right-click the chart and select *Visible Items»X Scrollbar* from the shortcut menu to display a scroll bar.

### 7.13.2 Configuring Chart Update Modes

You can configure how the chart updates to display new data. Right-click the chart and select *Advanced»Update Mode* from the shortcut menu to set the chart update mode. The chart uses the following modes as shown in Figure 7.17 to display data:

**Strip Chart**—Shows running data continuously scrolling from left to right across the chart with old data on the left and new data on the right. A strip chart is similar to a paper tape strip chart recorder. The *strip chart* is the default update mode.

**Scope Chart**—Shows one item of data, such as a pulse or wave, scrolling partway across the chart from left to right. For each new value, the chart plots the value to the right of the last value. When the plot reaches the right border of the plotting area, LabVIEW erases the plot and begins plotting again from the left border. The retracing display of a scope chart is similar to an oscilloscope.

**Sweep Chart**—Works similarly to a scope chart except it shows the old data on the right and the new data on the left separated by a vertical line. LabVIEW does not erase the plot in a sweep chart when the plot reaches the right border of the plotting area. A sweep chart is similar to an Electrocardiogram (EKG) display.

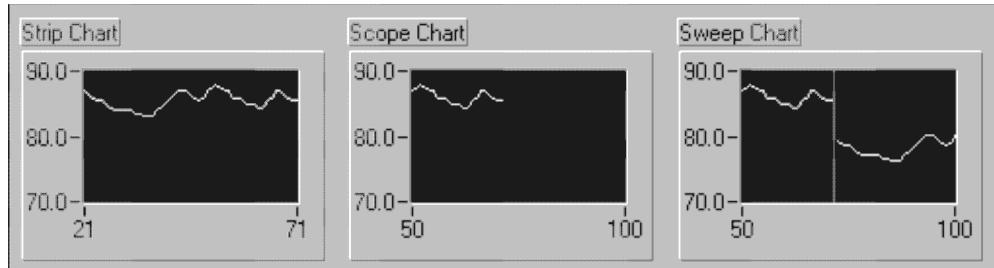
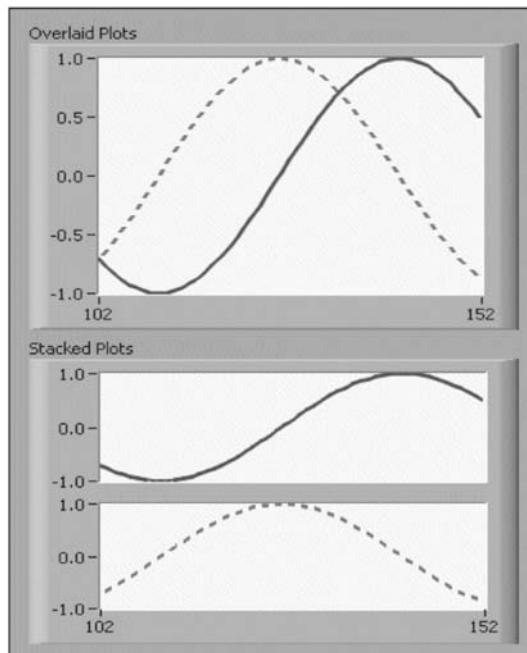


Figure 7.17 Example of each chart update mode.

The scope chart and sweep chart have retracing displays similar to an oscilloscope. Because retracing a plot requires less overhead, the scope chart and the sweep chart display plots significantly faster than the strip chart.

### 7.13.3 Using Overlaid and Stacked Plots

You can display multiple plots on a chart by using a single vertical scale, called *overlaid plots*, or by using multiple vertical scales, called *stacked plots*. Figure 7.18 shows examples of overlaid plots and stacked plots. Right-click the chart and select *Stack Plots* from the shortcut menu to view the chart plots as multiple vertical scales. Right-click the chart and select *Overlay Plots* to view the chart plots as a single vertical scale.



**Figure 7.18** Overlaid plots and stacked plots.

## 7.14 DYNAMICALLY FORMATTING WAVEFORM GRAPHS

Wire a dynamic data type output to a waveform graph to automatically format the plot legend and *x*-scale time stamp for the graph. For example, if you configure the Simulate Signal Express VI to generate a sine wave and to use absolute time and wire the output of the Simulate Signal Express VI to a waveform graph, the plot legend of the graph automatically updates the plot label to sine, and the *x*-scale displays the time and date when you run the VI. Right-click the graph and select *Ignore Attributes* from the shortcut menu to ignore the plot legend label the dynamic data includes. Right-click the graph and select *Ignore Time Stamp* from the shortcut menu to ignore the time stamp configuration the dynamic data includes.

## 7.15 CONFIGURING A GRAPH OR CHART

Right-click a graph or chart and select one of the following shortcut menu options to configure a graph or chart.

**Stack Plots**—Stacks the plots. Remove the checkmark from this shortcut menu option to overlay the plots. This option is available only for charts.

**Chart History Length**—Launches the *Chart History Length* dialog box. This option is available only for charts.

**Transpose Array**—Switches the *x*- and *y*-data before plotting. You also can use the *Transpose Array* property to switch the *x*- and *y*-data before plotting programmatically.

**Autosize Legend**—Automatically resizes the plot legend to the width of the longest plot name visible in the legend.

You also can configure the *x*- and *y*-axes on a graph or chart.

## 7.16 DISPLAYING SPECIAL PLANES ON THE XY GRAPH

You can display Nyquist planes, Nichols planes, S planes and Z planes on the *XY* graph.

### 7.16.1 Displaying a Nyquist Plane

Complete the following steps to display a Nyquist plane.

**Step 1:** Place an *XY* graph on the front panel.

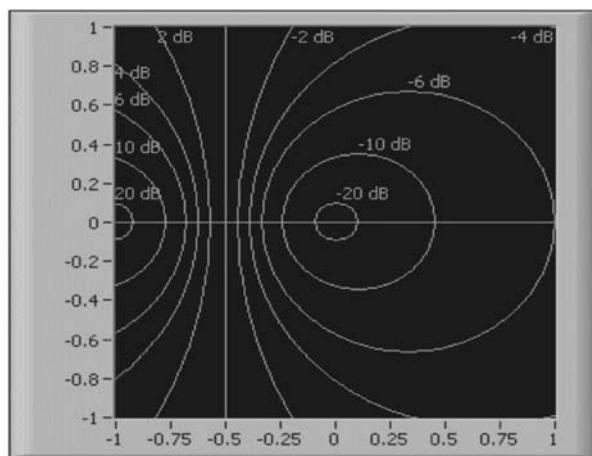
**Step 2:** Double-click the minimum and maximum values on the *x*- and *y*-scale and change both minimum values to  $-1$  and both maximum values to  $1$ .

**Step 3:** Right-click the *XY* graph and select *Optional Plane»Nyquist* from the shortcut menu.

**Step 4:** (Optional) To view the Nyquist plane without labels, right-click the *XY* graph and remove the checkmark from the *Optional Plane»Show Nyquist Labels* shortcut menu item.

**Step 5:** (Optional) To view the Nyquist plane without the Cartesian lines, right-click the *XY* graph and remove the checkmark from the *Optional Plane»Show Cartesian Lines* shortcut menu item.

To remove the Nyquist plane, right-click the *XY* graph and select *Optional Plane»None* from the shortcut menu. Figure 7.19 shows a Nyquist plane without the Cartesian lines.



**Figure 7.19** Nyquist plane without the Cartesian lines.

### 7.16.2 Displaying a Nichols Plane

Complete the following steps to display a Nichols plane.

**Step 1:** Place an XY graph on the front panel.

**Step 2:** Double-click the minimum and maximum values on the *x*-scale and change the minimum value to 0 and the maximum value to 360.

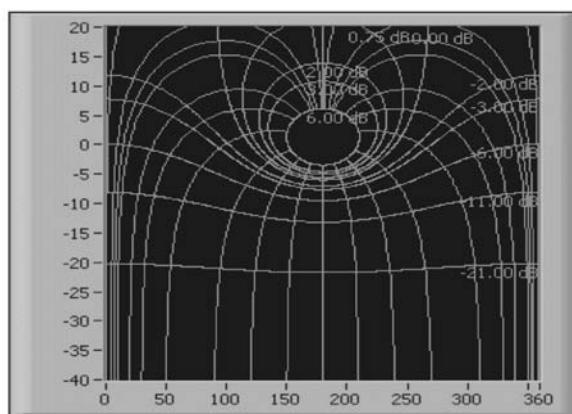
**Step 3:** Double-click the minimum and maximum values on the *y*-scale and change the minimum value to -40 and the maximum value to 20.

**Step 4:** Right-click the XY graph and select *Optional Plane»Nichols* from the shortcut menu.

**Step 5:** (Optional) To view the Nichols plane without labels, right-click the XY graph and remove the checkmark from the *Optional Plane»Show Nichols Labels* shortcut menu item.

**Step 6:** (Optional) To view the Nichols plane without the Cartesian lines, right-click the XY graph and remove the checkmark from the *Optional Plane»Show Cartesian Lines* shortcut menu item.

To remove the Nichols plane, right-click the XY graph and select *Optional Plane»None* from the shortcut menu. Figure 7.20 shows a Nichols plane without the Cartesian lines.



**Figure 7.20** Nichols plane without the Cartesian lines.

### 7.16.3 Displaying an S Plane

Complete the following steps to display an S plane.

**Step 1:** Place an XY graph on the front panel.

**Step 2:** Double-click the minimum and maximum values on the *x*-scale and change the minimum value to -1 and the maximum value to 0.

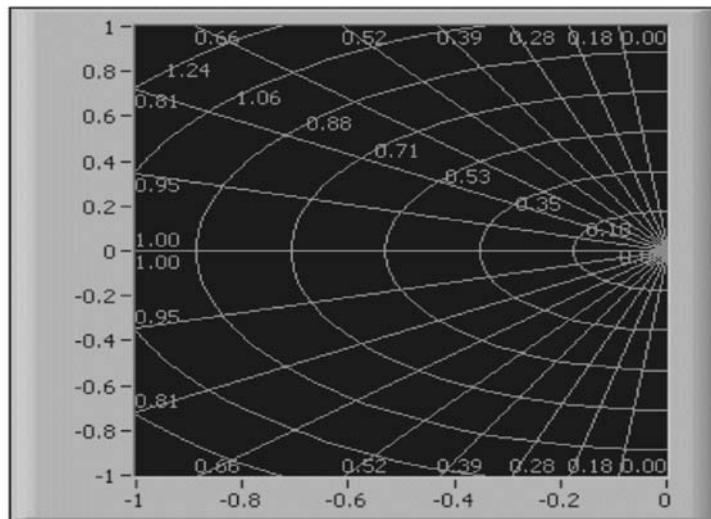
**Step 3:** Double-click the minimum and maximum values on the *y*-scale and change the minimum value to -1 and the maximum value to 1.

**Step 4:** Right-click the XY graph and select *Optional Plane»S Plane* from the shortcut menu.

**Step 5:** (Optional) To view the S plane without labels, right-click the XY graph and remove the checkmark from the *Optional Plane»Show S Plane Labels* shortcut menu item.

**Step 6:** (Optional) To view the S plane without the Cartesian lines, right-click the XY graph and remove the checkmark from the *Optional Plane»Show Cartesian Lines* shortcut menu item.

To remove the S plane, right-click the XY graph and select *Optional Plane»None* from the shortcut menu. Figure 7.21 shows an S plane without the Cartesian lines.



**Figure 7.21** S plane without the Cartesian lines.

#### 7.16.4 Displaying a Z Plane

Complete the following steps to display a Z plane.

**Step 1:** Place an XY graph on the front panel.

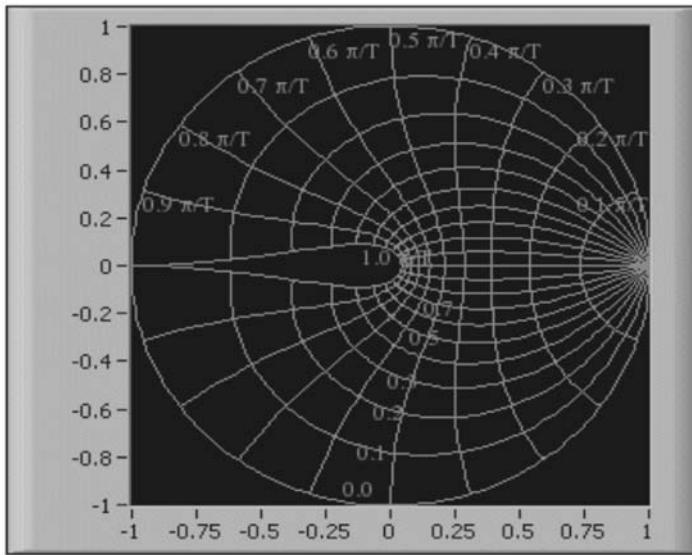
**Step 2:** Double-click the minimum and maximum values on the x- and y-scale and change both minimum values to -1 and both maximum values to 1.

**Step 3:** Right-click the XY graph and select *Optional Plane»Z Plane* from the shortcut menu.

**Step 4:** (Optional) To view the Z plane without labels, right-click the XY graph and remove the checkmark from the *Optional Plane»Show Z Plane Labels* shortcut menu item.

**Step 5:** (Optional) To view the Z plane without the Cartesian lines, right-click the XY graph and remove the checkmark from the *Optional Plane»Show Cartesian Lines* shortcut menu item.

To remove the Z plane, right-click the XY graph and select *Optional Plane»None* from the shortcut menu. Figure 7.22 shows a Z plane without the Cartesian lines.



**Figure 7.22** Z plane without the Cartesian lines.

## SUMMARY

- The waveform chart is a special numeric indicator that displays one or more plots.
- The waveform chart has the following three update modes:
  - ◆ A strip chart shows running data continuously scrolling from left to right across the chart.
  - ◆ A scope chart shows one item of data, such as a pulse or wave, scrolling partway across the chart from left to the right.
  - ◆ A sweep display is similar to an EKG display. A sweep works similarly to a scope except it shows the old data on the right and the new data on the left separated by a vertical line.
- Waveform graphs and XY graphs display data from arrays.
- Right-click a waveform chart or graph or its components to set attributes of the chart and its plots.
- You can display more than one plot on a graph using the *Build Array* function located on the *Functions»All Functions»Array* palette and the *Bundle* function located on the *Functions»All Functions»Cluster* palette for charts and XY graphs. The graph becomes a multiplot graph when you wire the array of outputs to the terminal.
- You can use intensity charts and graphs to plot three-dimensional data. The third dimension is represented by different colors corresponding to a color mapping that you define. Intensity charts and graphs are commonly used in conjunction with spectrum analysis, temperature display, and image processing.
- When you wire data to charts and graphs, use the *Context Help* window to determine how to wire them.
- You can display Nyquist planes, Nichols planes, S planes, and Z planes on the XY graph.

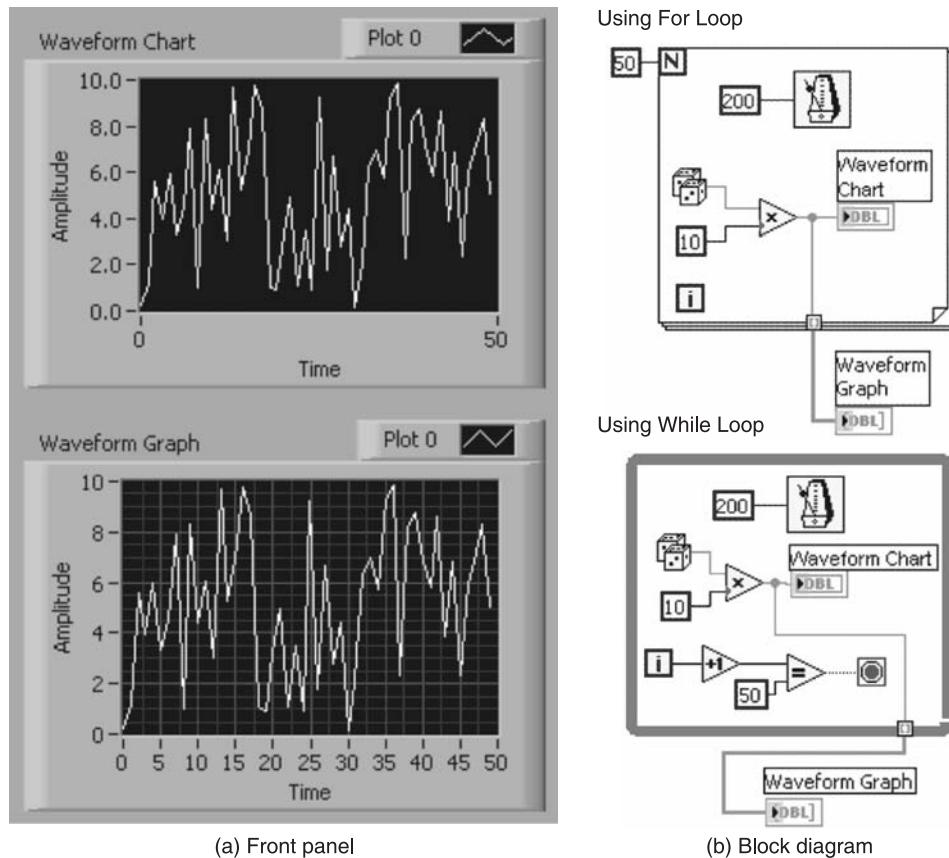
---

## MISCELLANEOUS SOLVED PROBLEMS

---

**Problem 7.1** Build a VI that generates 50 random numbers and plot it on a waveform chart using For and While Loops. Accumulate the random numbers into an array and display it on waveform graph.

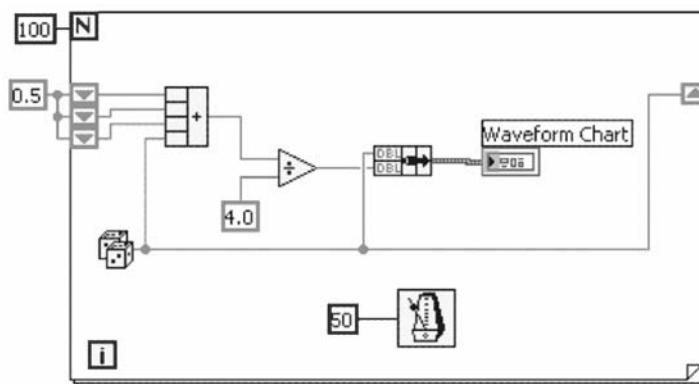
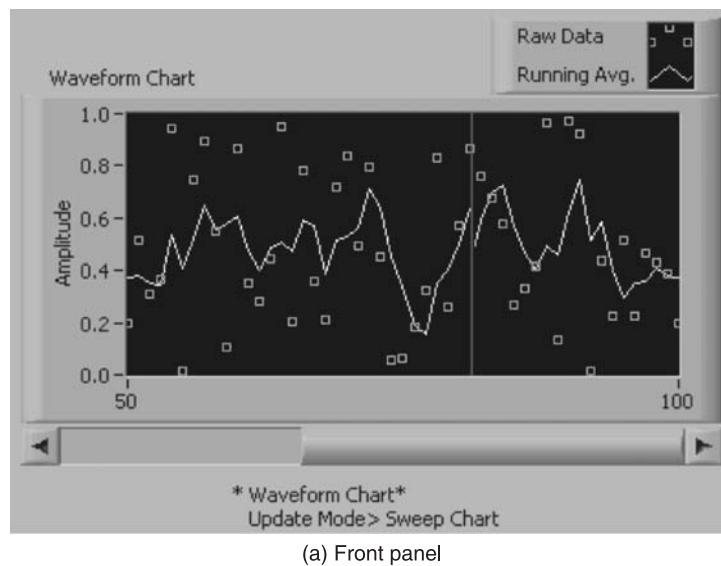
**Solution** To solve the waveform chart problem using For and While Loops create the front panel and the block diagram as shown in Figures P7.1(a) and P7.1(b).



**Figure P7.1**

**Problem 7.2** Build a VI that displays two random plots on a waveform chart in sweep update mode. The plots should be a random plot and a running average of the last four points.

**Solution** Create the front panel and the block diagram as shown in Figures P7.2(a) and P7.2(b) to solve the problem.

**Figure P7.2**

**Problem 7.3** Build a VI to generate sine waveform with options to vary the amplitude, number of waves, offset,  $t_0$ , number of points in a waveform and  $\Delta t$ .

**Solution** To solve the problem create the front panel and the block diagram as shown in Figure P7.3(a) and P7.3(b).

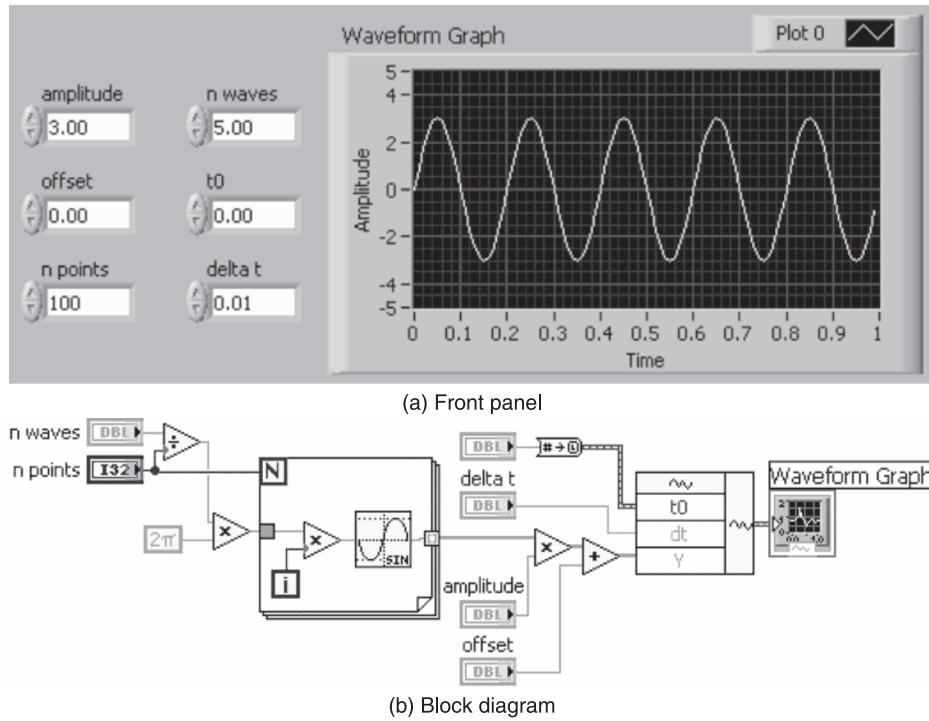
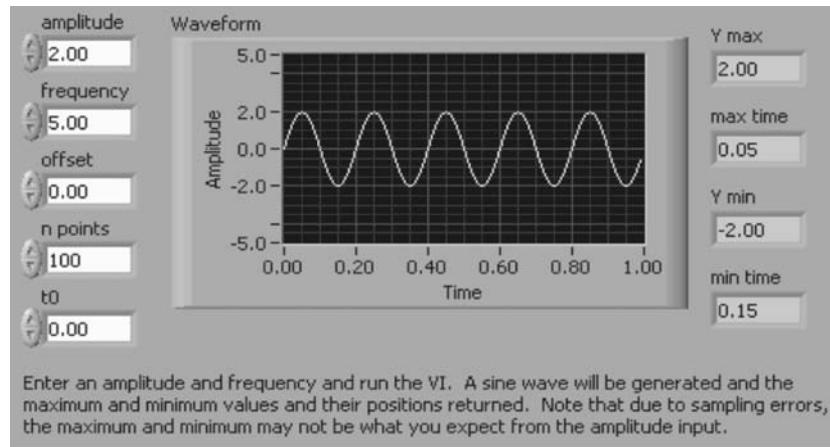


Figure P7.3

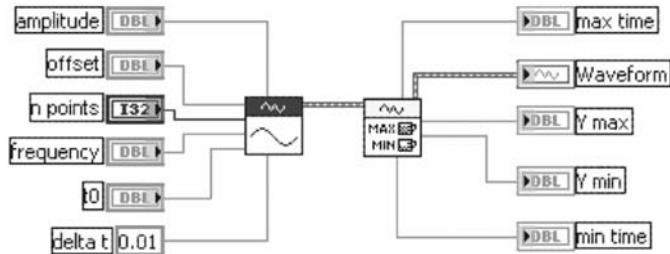
**Problem 7.4** Using the above VI as sub VI find the maximum and minimum values of the waveform and the time at which the waveform is maximum and minimum.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P7.4(a) and P7.4(b).



(a) Front panel

Figure P7.4 (Contd.)

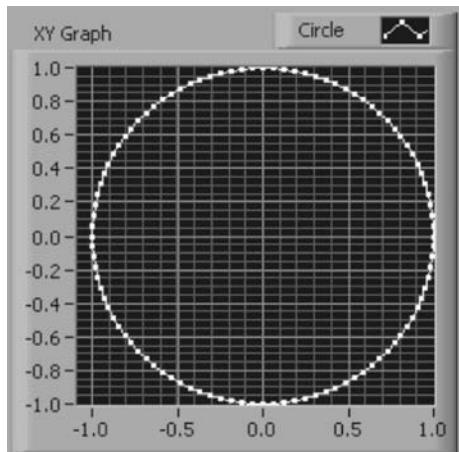


(b) Block diagram

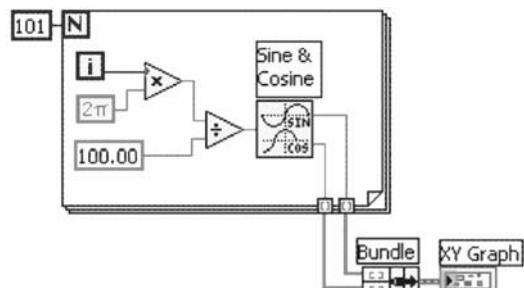
**Figure P7.4**

**Problem 7.5** Build a VI to plot a circle in the XY graph using a For Loop.

**Solution** To plot a circle in the XY graph using a For Loop, create the front panel and the block diagram as shown in Figures P7.5(a) and P7.5(b).



(a) Front panel

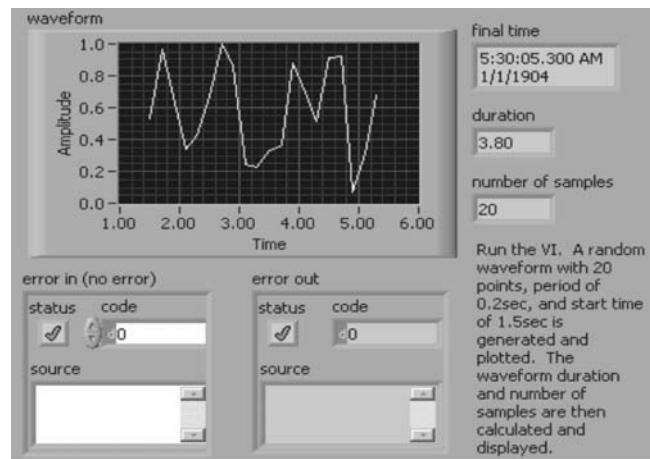


(b) Block diagram

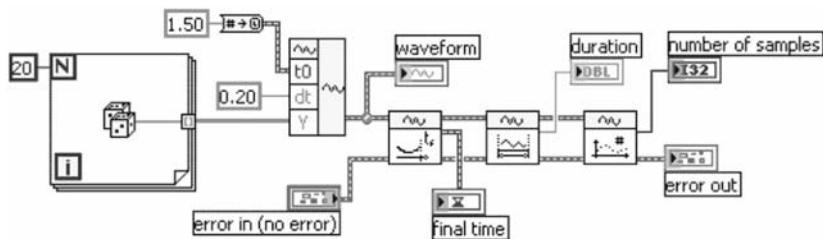
**Figure P7.5**

**Problem 7.6** Build a VI to generate and plot a waveform with 20 points, period of 0.2 sec, and a start time of 1.5 sec. Display total duration taken for the entire plot, number of samples plotted and the final time (time at which final data is plotted).

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P7.6(a) and P7.6(b).



(a) Front panel

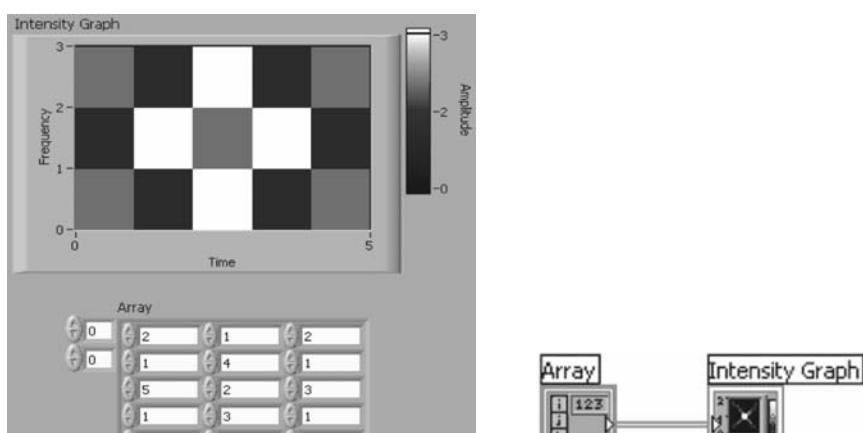


(b) Block diagram

**Figure P7.6**

**Problem 7.7** Build a VI to plot different colors in an intensity graph using an array.

**Solution** To create an intensity graph using an array, build the front panel and the block diagram as shown in Figures P7.7(a) and P7.7(b).



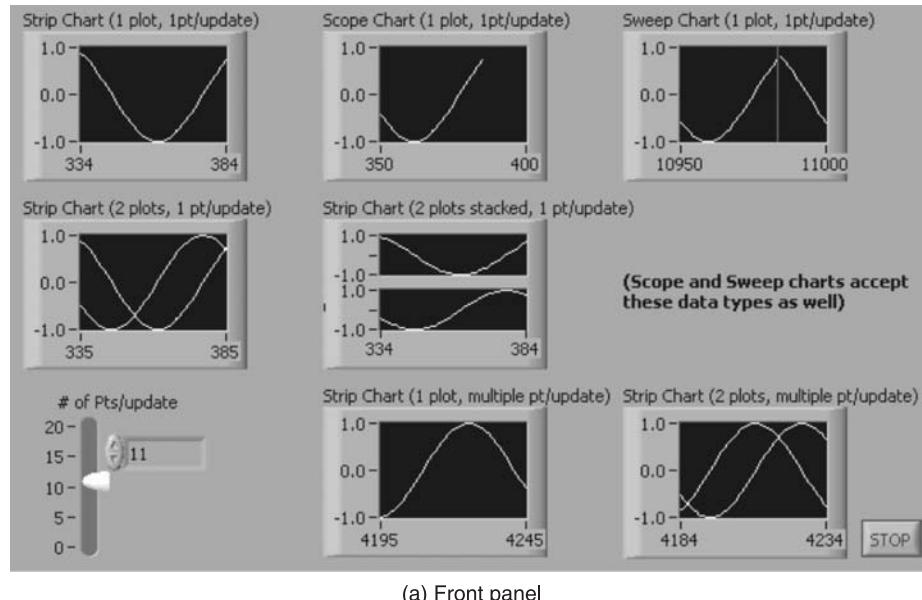
(a) Front panel

(b) Block diagram

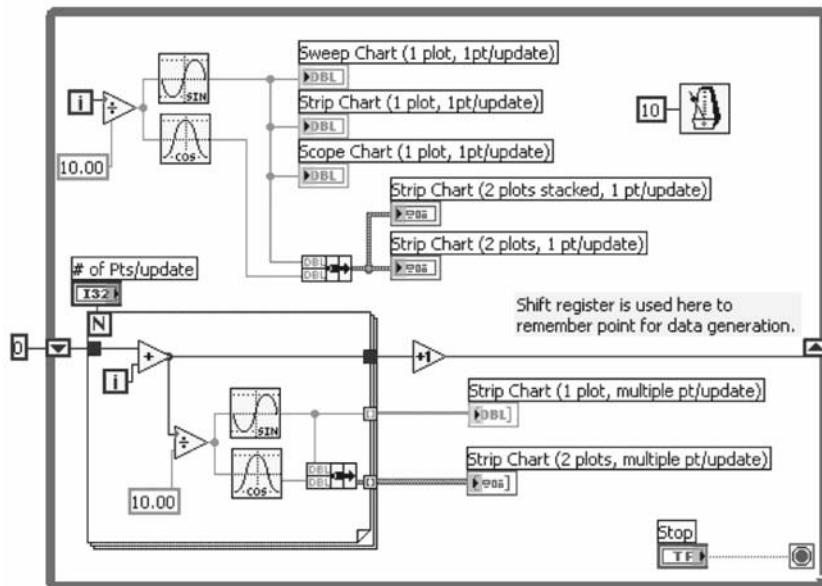
**Figure P7.7**

**Problem 7.8** Build a VI to examine the differences charts types like Strip Chart, Scope Chart and Sweep Chart.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P7.8(a) and P7.8(b).



(a) Front panel



(b) Block diagram

**Figure P7.8**

---

## REVIEW QUESTIONS

---

1. Draw and explain the various types of graphs and charts in LabVIEW.
2. What is the basic difference between a waveform chart and a graph?
3. Draw and explain how to get single and multiple plots on Waveform Charts.
4. What are the three update modes of waveform chart and list their practical applications?
5. What are the special features of an XY plot and what are its applications?
6. Draw and explain the concept of the intensity chart operation.
7. How are digital waveform graphs used to display digital data, especially when you work with timing diagrams or logic analyzers?
8. List the different types of 3D graphs.
9. Draw and explain the importance of the basic elements of a graph.

---

## EXERCISES

---

1. Build a VI to generate a sine wave using Simulate Signal Express VI. Change wave parameters like amplitude and frequency and plot it in a waveform graph.
2. Generate the sine wave of different amplitudes. Using Scaling and Mapping Express VI do normalization, linearization and interpolation of the generated waveform. Display both original waveform and modified waveform in a Waveform Graph.
3. Build a VI to generate a waveform using Simulate Signal Express VI. Using Amplitude and Level Measurements Express VI find the peak-to-peak value, RMS value, mean, negative peak and positive peak of the generated waveform.
4. Build a VI to plot random numbers in a waveform chart. Indicate under range and over range. Provide an option to programmatically change the update modes using a property node (strip, scope and sweep chart modes).
5. Compute the equations for  $X$  ranging from 0 to 10 in steps of 0.2 and plot them on a suitable waveform indicator. Find out whether they intersect. If so, what are the intersect ( $X$  and  $Y$ ) coordinates of the intersect?

$$Y_1 = X^2 + 2X + 1$$

$$Y_2 = X^2 + 6X - 3$$

6. Build a VI to generate two waveforms of different amplitude and frequency. Add the signals to find the resultant waveform and plot it on a separate Waveform Graph.
7. Build a VI that continuously measures the temperature once per second and displays the temperature on a scope chart. If the temperature goes above or below limits specified with front panel controls, the VI turns on a front panel LED. The chart plots the temperature and the upper and lower temperature limits.



# STRUCTURES

---

## 8.1 INTRODUCTION

Structures are graphical representations of the loops and case statements of text-based programming languages. There are cases when a decision must be made in a program. In text-based programs, this can be accomplished with statements like *if-else*, *case* and so on. LabVIEW includes many different ways of making decisions. The simplest of these methods is the select function located in the functions palette. This function selects between two values dependent on a Boolean input.

Use structures on the block diagram to repeat blocks of code and to execute code conditionally or in a specific order. Like other nodes, structures have terminals that connect them to other block diagram nodes, execute automatically when input data are available, and supply data to output wires when execution is complete. Each structure has a distinctive, resizable border to enclose the section of the block diagram that executes according to the rules of the structure. The section of the block diagram inside the structure border is called a *subdiagram*. The terminals that feed data into and out of structures are called *tunnels*. A tunnel is a connection point on a structure border.

Use the following structures located on the *Structures* palette to control how a block diagram executes processes:

- **For Loop**—Executes a subdiagram a set number of times.
- **While Loop**—Executes a subdiagram until a condition occurs.
- **Case structure**—Contains multiple subdiagrams, only one of which executes depending on the input value passed to the structure.

- **Sequence structure**—Contains one or more subdiagrams that execute in sequential order.
- **Event structure**—Contains one or more subdiagrams that execute depending on how the user interacts with the VI.
- **Timed Structures**—Execute one or more subdiagrams with time bounds and delays.
- **Diagram Disable Structure**—Has one or more subdiagrams, or cases, of which only the enabled subdiagram executes.
- **Conditional Disable Structure**—Has one or more subdiagrams, or cases, exactly one of which LabVIEW uses for the duration of execution, depending on the configuration.

The two most commonly used For Loop and While Loop are explained in detail in Chapter 4.

## 8.2 CASE STRUCTURES

A case structure executes one subdiagram depending on the input value passed to the structure.

Complete the following steps to create a *Case* structure.

**Step 1:** Place a *Case* structure on the block diagram.

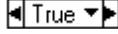
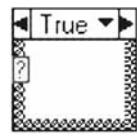
**Step 2:** Wire an input value to the selector terminal to determine which case executes. You must wire an integer, Boolean value, string, or enumerated type value to the selector terminal. You also can wire an error cluster to the selector terminal to handle errors.

**Step 3:** Place objects inside the *Case* structure to create subdiagrams that the *Case* structure can execute. If necessary, add or duplicate subdiagrams. If the data type of the selector terminal is Boolean, the structure has a TRUE case and a FALSE case. If the selector terminal is an integer, string, or enumerated type value, the structure can have any number of cases.

**Step 4:** For each case, use the *Labeling* tool to enter a single value or lists and ranges of values in the case selector label at the top of the *Case* structure. For lists, use commas to separate values. For numeric ranges, specify a range as 10.20, meaning all numbers from 10 to 20 inclusively. If necessary (optional), specify a default case.

A *Case* structure, shown, has two or more subdiagrams, or cases.

Only one subdiagram is visible at a time, and the structure executes only one case at a time. An input value determines which subdiagram executes. The *Case* structure is similar to *switch* statements or *if...then...else* statements in text-based programming languages. The case selector label at the top of the *Case* structure, shown, contains the name of the selector value that corresponds to the case in the center and decrement and increment arrows on each side.



Click the decrement and increment arrows to scroll through the available cases. You also can click the down arrow next to the case name and select a case from the pull-down menu.

Wire an input value, or selector, to the selector terminal, shown, to determine which case executes.

You must wire an integer, Boolean value, string, or enumerated type value to the selector terminal. You can position the selector terminal anywhere on the left border of the *Case* structure.

If the data type of the selector terminal is Boolean, the structure has a TRUE case and a FALSE case. If the selector terminal is an integer, string, or enumerated type value, the structure can have any number of cases.

Specify a default case for the *Case* structure to handle out-of-range values. Otherwise, you must explicitly list every possible input value. For example, if the selector is an integer and you specify cases for 1, 2 and 3, you must specify a default case to execute if the input value is 4 or any other unspecified integer value.

### 8.2.1 Case Selector Values and Data Types

You can enter a single value or lists and ranges of values in the case selector label. For lists, use commas to separate values. For numeric ranges, specify a range as 10...20, meaning all numbers from 10 to 20 inclusively. You also can use open-ended ranges. For example, ... 100 represents all numbers less than or equal to 100, and 100.. represents all numbers greater than or equal to 100. You also can combine lists and ranges, for example ... 5, 6, 7 ... 10, 12, 13, 14. When you enter values that contain overlapping ranges in the same case selector label, the *Case* structure redisplays the label in a more compact form. The previous example redisplays as ... 10, 12..14. For string ranges, a range of *a* ... *c* includes all of *a* and *b*, but not *c*. A range of *a* ... *c*, *c* includes the ending value of *c*.

When you enter string and enumerated values in a case selector label, the values display in quotation marks, for example “red”, “green” and “blue”. However, you do not need to type the quotation marks when you enter the values unless the string or enumerated value contains a comma or range symbol (“,” or “..”). In a string value, use special backslash codes for non-alphanumeric characters, such as \r for a carriage return, \n for a line feed, and \t for a tab.

If you change the data type of the wire connected to the selector terminal of a *Case* structure, the *Case* structure automatically converts the case selector values to the new data type when possible. If you convert a string to a numeric value, LabVIEW converts only those string values that represent a number. The other values remain strings. If you convert a number to a Boolean value, LabVIEW converts 0 to FALSE and 1 to TRUE, and all other numeric values become strings.

If you enter a selector value that is not the same type as the object wired to the selector terminal, the value appears red to indicate that you must delete or edit the value before the structure can execute, and the VI will not run. Also, because of the possible round-off error inherent in floating-point arithmetic, you cannot use floating-point numbers as case selector values. If you wire a floating-point value to the case, LabVIEW rounds the value to the nearest even integer. If you type a floating-point value in the case selector label, the value appears red to indicate that you must delete or edit the value before the structure can execute.

### 8.2.2 Input and Output Tunnels

You can create multiple input and output tunnels for a *Case* structure. Inputs are available to all cases, but cases do not have to use each input. However, you must define each output tunnel for each case. When you create an output tunnel in one case, tunnels appear at the same position on the border in all the other cases. If even one output tunnel is not wired, all output tunnels on the structure appear as white squares. You can define a different data source for the same output tunnel in each case, but the data types must be compatible for each case. You also can right-click the

output tunnel and select *Use Default If Unwired* from the shortcut menu to use the default value for the tunnel data type for all unwired tunnels.

### 8.2.3 Using Case Structures for Error Handling

When you wire an error cluster to the selector terminal of a *Case* structure, the case selector label displays two cases—Error and No Error—and the border of the *Case* structure changes color—red for Error and green for No Error. If an error occurs, the *Case* structure executes the *Error* subdiagram.

## 8.3 SEQUENCE STRUCTURES

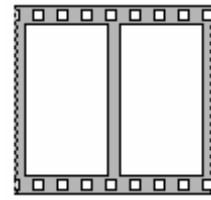
A sequence structure contains one or more subdiagrams, or frames, that execute in sequential order. Within each frame of a sequence structure, as in the rest of the block diagram, data dependency determines the execution order of nodes. Sequence structures are not used commonly in LabVIEW. Use the sequence structures to control the execution order when natural data dependency does not exist and flow-through parameters are not available. There are two types of sequence structures—the *Flat Sequence* structure and the *Stacked Sequence* structure.

### 8.3.1 Flat Sequence Structure

The *Flat Sequence* structure, shown as follows, displays all the frames at once and executes the frames from left to right and when all data values wired to a frame are available, until the last frame executes. The data values leave each frame as the frame finishes executing.

Use the *Flat Sequence* structure to avoid using sequence locals and to better document the block diagram. When you add or delete frames in a *Flat Sequence* structure, the structure resizes automatically.

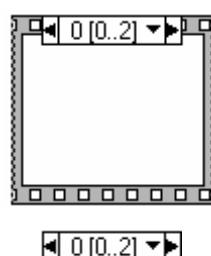
To convert a *Flat Sequence* structure to a *Stacked Sequence* structure, right-click the *Flat Sequence* structure and select *Replace with Stacked Sequence* from the shortcut menu. If you change a *Flat Sequence* to a *Stacked Sequence* and then back to a *Flat Sequence*, LabVIEW moves all input terminals to the first frame of the sequence. The final *Flat Sequence* should operate the same as the *Stacked Sequence*. After you change the *Stacked Sequence* to a *Flat Sequence* with all input terminals on the first frame, you can move wires to where they were located in the original *Flat Sequence*.



### 8.3.2 Stacked Sequence Structure

The *Stacked Sequence* structure, shown as follows, stacks each frame so you see only one frame at a time and executes frame 0, then frame 1, and so on until the last frame executes.

The *Stacked Sequence* structure returns data only after the last frame executes. Use the *Stacked Sequence* structure if you want to conserve space on the block diagram. To convert a *Stacked Sequence* structure to a *Flat Sequence* structure, right-click the *Stacked Sequence* structure and select *Replace»Replace with Flat Sequence* from the shortcut menu. The sequence



selector identifier, shown as follows, at the top of the *Stacked Sequence* structure contains the current frame number and range of frames.

Use the sequence selector identifier to navigate through the available frames and rearrange frames. The frame label in a *Stacked Sequence* structure is similar to the case selector label of the *Case* structure. The frame label contains the frame number in the center and decrement and increment arrows on each side. Click the decrement and increment arrows to scroll through the available frames. You also can click the down arrow next to the frame number and select a frame from the pull-down menu. Right-click the border of a frame, select *Make This Frame*, and select a frame number from the shortcut menu to rearrange the order of a *Stacked Sequence* structure. Unlike the case selector label, you cannot enter values in the frame label. When you add, delete, or rearrange frames in a *Stacked Sequence* structure, LabVIEW automatically adjusts the numbers in the frame labels.

To pass data from one frame to any subsequent frame of a *Stacked Sequence* structure, use a sequence local terminal shown .

An outward-pointing arrow appears in the sequence local terminal of the frame that contains the data source. The terminal in subsequent frames contains an inward-pointing arrow, indicating that the terminal is a data source for that frame. You cannot use the sequence local terminal in frames that precede the first frame where you wired the sequence local.

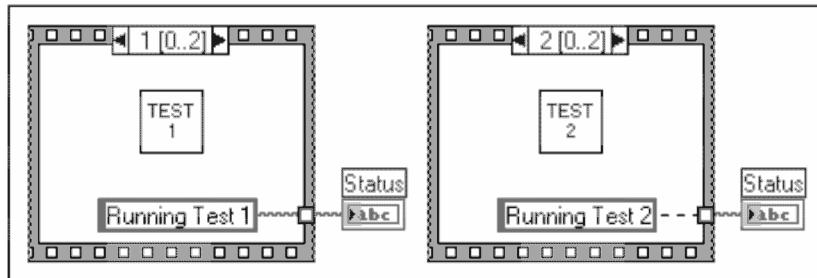
### 8.3.3 Using Sequence Structures

The tunnels of a sequence structure can have only one data source, unlike *Case* structures. The output can emit from any frame. If you use a *Flat Sequence* structure, the data from outside the sequence structure enter the frame as each frame executes. The data leave the frame after the frame executes. If you use a *Stacked Sequence* structure, the structure does not start to execute until all data wired to the structure arrive. The data wired from each frame leave only when all the frames complete execution. As with *Case* structures, data at input tunnels are available to all frames in either the *Flat Sequence* or the *Stacked Sequence* structure. To pass data from one frame to any subsequent frame in a *Flat Sequence* structure, wire from the tunnel of one frame to any other frame in the structure. You can wire through a frame or out of a one frame to another frame.

### 8.3.4 Avoiding Overuse of Sequence Structures

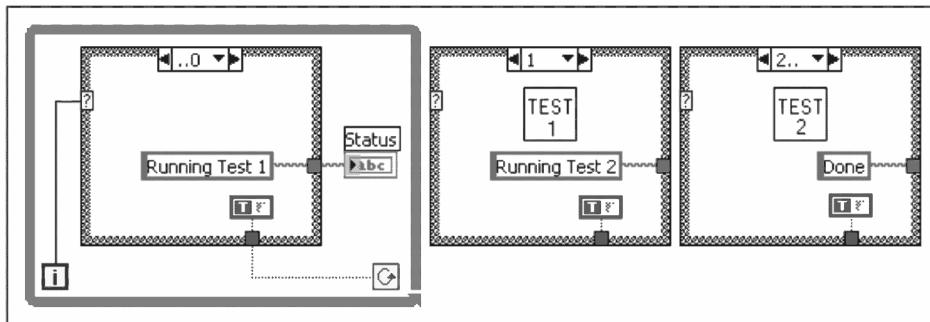
To take advantage of the inherent parallelism in LabVIEW, avoid overusing sequence structures. Sequence structures guarantee the order of execution and prohibit parallel operations. For example, asynchronous tasks that use I/O devices, such as PXI, GPIB, serial ports and DAQ devices, can run concurrently with other operations if sequence structures do not prevent them from doing so.

When you need to control the execution order, consider establishing data dependency between the nodes. For example, you can use flow-through parameters such as error I/O to control the execution order. Also, do not use sequence structures to update an indicator from multiple frames of the sequence structure. For example, a VI used in a test application might have a *Status* indicator that displays the name of the current test in progress. If each test is a subVI called from a different frame, you cannot update the indicator from each frame as shown in Figure 8.1 by the broken wire in the *Stacked Sequence* structure.



**Figure 8.1** Stacked Sequence structure.

Because all frames of a *Stacked Sequence* structure execute before any data pass out of the structure, only one frame can assign a value to the *Status* indicator. Instead, use a *Case* structure and a While Loop as shown in Figure 8.2.



**Figure 8.2** Use of Case structure and a While Loop.

Each case in the *Case* structure is equivalent to a sequence structure frame. Each iteration of the While Loop executes the next case. The *Status* indicator displays the status of the VI for each case. The *Status* indicator is updated in the case before the one that calls the corresponding subVI because data pass out of the structure after each case executes. Unlike a sequence structure, a *Case* structure can pass data to end the While Loop during any case. For example, if an error occurs while running the first test, the *Case* structure can pass FALSE to the conditional terminal to end the loop. However, a sequence structure must execute all its frames even if an error occurs.

### 8.3.5 Adding and Removing Sequence Local Terminals

Use a sequence local terminal to pass data from one frame to any subsequent frame in a *Stacked Sequence* structure. Complete the following steps to add a sequence local terminal.

**Step 1:** Move to the frame from which you want to pass data to a subsequent frame.

**Step 2:** On the right border of the structure, right-click the structure border and select *Add Sequence Local* from the shortcut menu.

**Step 3:** Wire the data you want to pass to the sequence local terminal. An outward-pointing arrow appears in the terminal.

**Step 4:** In subsequent frames, wire the sequence local terminal to any terminals where you want to pass the data from the previous frame. The terminals in subsequent frames contain an inward-pointing arrow, indicating that the terminal is a data source for that frame.

To remove a sequence local terminal, right-click the sequence local terminal and select *Remove* from the shortcut menu.

## 8.4 CUSTOMIZING STRUCTURES

### 8.4.1 Placing Structures on the Block Diagram

Complete the following steps to place a structure on the block diagram.

**Step 1:** Select a structure on the *Structures* palette. The cursor becomes a small icon of the structure.

**Step 2:** Click the block diagram where you want to place the top corner of the structure and move the cursor down and to the right or left.

**Step 3:** Click the block diagram again when the structure is the size you want.

After you place the structure, you can resize it.

### 8.4.2 Placing Objects inside Structures

Place an object inside a structure by dragging it inside the structure or by building the structure around the object. You cannot put an object inside a structure by moving the structure over the object.

Complete the following steps to practice placing objects inside a structure.

**Step 1:** Place the *Case* structure on the block diagram.

**Step 2:** Place the *Tick Count (ms)* function inside the structure.

**Step 3:** Move the *Tick Count (ms)* function close to the border of the structure. Notice that when you place or move an object in a structure near the structure border, the structure resizes to add space for that object. To disable the automatic resizing behavior for a structure, right-click the structure border and select *Auto Grow* from the shortcut menu to remove the checkmark.

**Step 4:** Move the *Tick Count (ms)* function outside the structure.

**Step 5:** Place another structure around the *Tick Count (ms)* function.

**Step 6:** Select the second structure and delete it. Notice that you also deleted the function inside the structure.

**Step 7:** Place another *Tick Count (ms)* function outside the remaining structure.

**Step 8:** Move the structure over the *Tick Count (ms)* function. The shadow on the function indicates that the function is on top of the structure but is not inside it.

**Step 9:** Select the *Case* structure and delete it. Notice that you did not delete the function because it was not inside the structure.

#### 8.4.3 Removing Structures without Deleting Objects in the Structure

Complete the following steps to remove a structure without deleting the objects in the structure.

**Step 1:** Right-click the structure you want to remove.

**Step 2:** Select *Remove For Loop* from the shortcut menu or the similar option for any of the other structures.

With For Loops and While Loops, LabVIEW copies the contents of the loop to the block diagram and automatically connects any elements wired through tunnels.

Removing a *Case*, *Event*, *Flat Sequence*, or *Stacked Sequence* structure preserves only the current case or frame and any front panel terminals. All other cases or frames are deleted. LabVIEW warns you that you will lose hidden cases or frames and asks if you want to cancel the operation.

#### 8.4.4 Resizing Structures

Complete the following steps to resize a structure on the block diagram.

**Step 1:** Move the positioning tool over the structure border. Resizing handles appear at the corners of the structure and in the middle of each structure border.

**Step 2:** Move the cursor over a resizing handle to change the cursor to the resizing cursor.

**Step 3:** Use the resizing cursor to drag the resizing handles until the dashed border outlines the size you want.

**Step 4:** Release the mouse button. The structure reappears in its new size.

When resizing a single frame in a *Flat Sequence* structure using the resizing handle on a shared edge of two frames, make sure you are resizing the correct frame by noting which frame displays resizing handles. To shrink a frame using the resizing handle on the left edge of a frame, drag the handle to the right. To grow a frame using the same resizing handle, drag the handle to the left. When you shrink or grow a frame, the other frames in the *Flat Sequence* structure remain their current size and move to adjust to the resized frame.

#### 8.4.5 Adding Cases to the Middle of an Ordered List

Complete the following steps to add a case to the middle of the order of a *Case* structure and shift existing cases down in the order.

**Step 1:** Add a new case at the end of the order.

**Step 2:** With the new case visible, right-click the structure border and select *Shift Diagram To Case* from the shortcut menu.

**Step 3:** Select the position for the new case. The subsequent subdiagrams shift down in the order.

You also can press the <Shift-Enter> keys in the case selector label to add a case to a *Case* structure.

#### 8.4.6 Adding, Duplicating and Deleting Subdiagrams

Right-click a *Case*, *Stacked Sequence*, *Flat Sequence*, *Conditional Disable*, or *Diagram Disable* structure border and select *Add Case* (or *Frame* or *Subdiagram*) *After* or *Before* from the shortcut menu to add a subdiagram.

Complete the following steps to make a copy of the visible subdiagram in a *Case*, *Stacked Sequence*, or *Conditional Disable* structure and insert it after the original subdiagram.

**Step 1:** Move to the subdiagram you want to duplicate.

**Step 2:** Right-click the structure border and select *Duplicate Case* (or *Frame* or *Subdiagram*) from the shortcut menu. The duplicate subdiagram contains all objects from the original subdiagram except front panel terminals.

**Step 3:** If you are configuring a *Conditional Disable* structure, select a *Symbol* and enter a *Value* in the *Configure Condition* dialog box.

When you add, delete or rearrange frames in a *Stacked Sequence* structure, LabVIEW automatically adjusts the numbers in the frame labels. When you add or delete frames in a *Flat Sequence* structure, the structure resizes automatically. You can rearrange the subdiagrams in a *Case* or *Conditional Disable* structure, but the order of the cases does not affect how the VI runs. You also can add or duplicate event cases.

To delete the visible subdiagram, right-click the structure border and select *Delete This Case* (or *Frame* or *Subdiagram*) from the shortcut menu. If only one subdiagram exists, you cannot delete the visible subdiagram.

### 8.5 TIMED STRUCTURES

Use timed structures on the block diagram to repeat blocks of code and to execute code in a specific order with time bounds and delays. Each timed structure has a distinctive, resizable border to enclose a section of the block diagram that executes according to the rules of the structure. The section of the block diagram inside the structure border is called a subdiagram. A timed structure has Input and Output nodes that feed data into and out of the structure to provide configuration data and return error and timing information. Timed structures can also have terminals on the structure border that feed data into and out of the structure subdiagrams. Use the following structures located on the *Timed Structures* palette to control how a block diagram executes with time bounds and delays:

- **Timed Loop**—Executes a subdiagram until a condition is met or interminably.
- **Timed Sequence**—Executes multiple subdiagrams in sequence.
- **Timed Loop with Frames**—Executes multiple subdiagrams in sequence until a condition is met or interminably. Add frames to a Timed Loop to create a Timed Loop with frames.

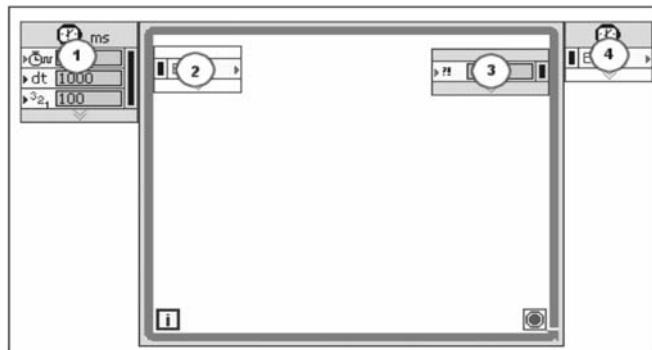
Use a Timed Loop and Timed Sequence to repeat a block of code and to execute code in a specific order with time bounds and delays. Use a Timed Loop with frames to repeatedly execute code in a specific order with time bounds and delays.

### 8.5.1 Timed Loop Structure

A Timed Loop executes a subdiagram, or frame, each iteration of the loop at the period you specify. Use the Timed Loop when you want to develop VIs with multi-rate timing capabilities, precise timing, feedback on loop execution and timing characteristics that change dynamically, or several levels of execution priority. Unlike the While Loop, the Timed Loop does not require wiring to the stop terminal. If you do not wire anything to the stop terminal, the loop will run interminably.

A Timed Loop executes below the time-critical priority of any VI but above high priority, which means that a Timed Loop executes in the data flow of a block diagram ahead of any VI not configured to run at a time-critical priority.

The Timed Loop includes (1) Input, (2) Left Data, (3) Right Data and (4) Output nodes as shown in Figure 8.3. By default, nodes of the Timed Loop do not display all of the available input and output terminals. You can resize nodes or right-click a node and use the shortcut menu to display hidden node terminals. You can set the initial configuration options of a Timed Loop by wiring values to the inputs of the Input node, or you can use the *Configure Timed Loop* dialog box, available by right-clicking the Input node and selecting *Configure Input Node* from the shortcut menu, to enter values for the options. Refer to the Configuring Timed Loops topic for more information about configuring a Timed Loop.



**Figure 8.3** Timed Loop.

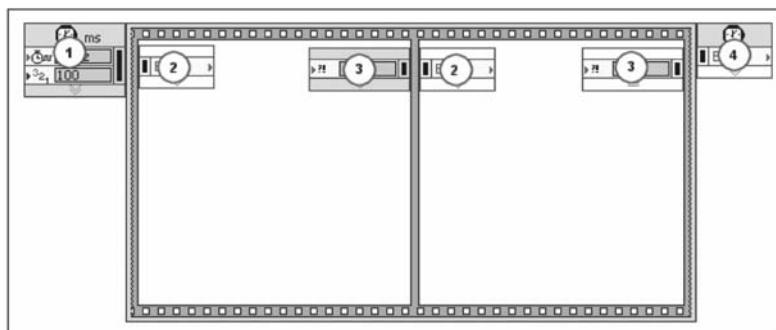
The *Left Data* node of the Timed Loop returns configuration option values and provides timing and status information about the previous loop iteration, such as if the iteration executed late, the time the iteration actually started, and when the iteration was expected to execute. You can wire values to the inputs of the *Right Data* node to configure the options of the next loop iteration dynamically, or you can use the *Configure Next Iteration* dialog box, available by right-clicking the *Right Data* node and selecting *Configure Input Node* from the shortcut menu, to enter values for the options.

The *Output* node returns error information received in the *Error in* input of the *Input* node, error information generated by the structure during execution, or error information from the task subdiagram that executes within the Timed Loop. The *Output* node also returns timing and status information.

### 8.5.2 Timed Sequence Structure

The timed sequence structure consists of one or more task subdiagrams, or frames, that execute sequentially and can be timed with an internal or external timing source. Use the Timed Sequence when you want to develop VIs with precise timing, execution feedback, timing characteristics that change dynamically, or several levels of execution priority. A Timed Sequence executes below the time-critical priority of any VI but above high priority, which means that a Timed Sequence executes in the data flow of a block diagram ahead of any VI not configured to run at a time-critical priority. Right-click the structure border to add, delete and merge frames.

The Timed Sequence includes (1) Input and (4) Output nodes, and (2) Left and (3) Right Data nodes for each frame as shown in Figure 8.4. By default, nodes of the Timed Sequence do not display all of the available input and output terminals. You can resize nodes or right-click a node and use the shortcut menu to display hidden terminals.



**Figure 8.4** Timed Sequence structure.

You can set the configuration options of the Timed Sequence by wiring values to terminals on the *Input* node, or you can use the *Configure Timed Sequence* dialog box, available by right-clicking the Input node and selecting *Configure Input Node* from the shortcut menu, to enter values for the options. Refer to the Configuring Timed Sequences topic for more information about configuring a *Timed Sequence*. The *Left Data* node of a *Timed Sequence* frame returns configuration option values and provides timing and status information about the current and previous frame, such as the expected start time, actual start time, and if the previous frame completed late. You can wire values to the *Right Data* node to configure the options of the next frame dynamically.

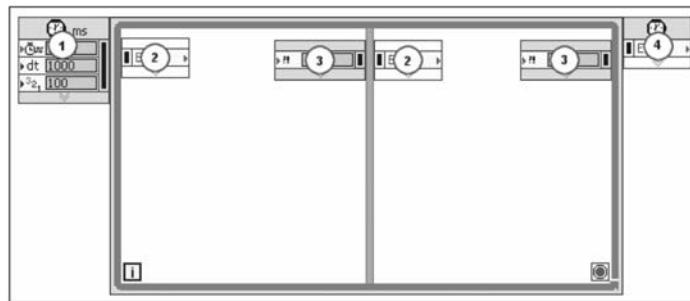
The *Output* node returns error information received in the *error in* input of the *Input* node, error information generated by the structure during execution, or error information from any task subdiagram that executes within a frame of the Timed Sequence. The *Output* node also returns timing and status information for the final frame.

### 8.5.3 Timed Loop with Frames Structure

You can add frames to a Timed Loop to execute multiple subdiagrams sequentially each iteration of the loop at the period you specify. A Timed Loop with frames behaves like a regular Timed Loop with an embedded sequence structure. Unlike the While Loop, the Timed Loop does not

require wiring to the stop terminal. If you do not wire anything to the stop terminal, the loop will run interminably.

Timed Loops execute below the time-critical priority of any VI but above high priority, which means that Timed Loops execute in the data flow of a block diagram ahead of any VI not configured to run at a time-critical priority. Right-click the structure border to add, delete and merge frames. The Timed Loop with frames includes (1) Input and (4) Output nodes, and (2) Left and (3) Right Data nodes for each frame as shown in Figure 8.5. By default, the nodes of the Timed Loop do not display all of the available input and output terminals. You can resize nodes or right-click a node and use the shortcut menu to display hidden terminals. You can set initial configuration options of the Timed Loop by wiring values to the inputs of the *Input* node, or you can use the *Configure Timed Loop With Frames* dialog box, available by right-clicking the *Input* node and selecting *Configure Input Node* from the shortcut menu, to enter values for the options. Refer to the Configuring Timed Loops topic for more information about configuring a Timed Loop with frames.



**Figure 8.5** Timed Loop with frames.

The *Left Data* node of a Timed Loop frame returns configuration option values and provides timing and status information about the previous loop iteration or frame. You can wire data to the *Right Data* node of a frame to configure the options of the next frame dynamically, or you can use the *Configure Next Frame Timing* dialog box, available by right-clicking the *Right Data* node and selecting *Configure Input node* from the shortcut menu, to enter values for the options. You also can wire data to the *Right Data* node of a frame to configure the options of the next iteration dynamically, or you can use the *Configure Next Iteration* dialog box, available by right-clicking the *Right Data* node of the last frame and selecting *Configure Input node* from the shortcut menu, to enter values for the options. The *Output* node returns error information received in the *Error in* input of the *Input* node, error information generated by the structure during execution, or error information from the task subdiagrams that execute within the Timed Loop frames. The *Output* node also returns timing and status information for the final frame.

## 8.6 FORMULA NODES

The *Formula Node* is a convenient text-based node you can use to perform mathematical operations on the block diagram. You do not have to access any external code or applications, and you do not have to wire low-level arithmetic functions to create equations. In addition to text-based equation expressions, the *Formula Node* can accept text-based versions of If statements, While loops, For

loops, and Do loops which are familiar to C programmers. These programming elements are similar to what you find in C programming but are not identical.

*Formula Nodes* are useful for equations that have many variables or are otherwise complicated and for using existing text-based code. You can copy and paste the existing text-based code into a *Formula Node* rather than recreating it graphically. *Formula Nodes* use type checking to make sure that array indexes are numeric data and that operands to the bit operations are integer data. *Formula Nodes* also check to make sure array indexes are in range. For arrays, an out-of-range value defaults to zero, and an out-of-range assignment defaults to nop to indicate no operation occurs. *Formula Nodes* also perform automatic type conversion.

### 8.6.1 Using the Formula Node

The *Formula Node*, shown in Figure 8.6, is a resizable box similar to the For Loop, While Loop, Case structure, Stacked Sequence structure and Flat Sequence structure. However, instead of containing a subdiagram, the *Formula Node* contains one or more C-like statements delimited by semicolons, as in the following example. As with C, add comments by enclosing them inside a slash/asterisk pair (*/\*comment\*/*) or by preceding them with two slashes (*//comment*).

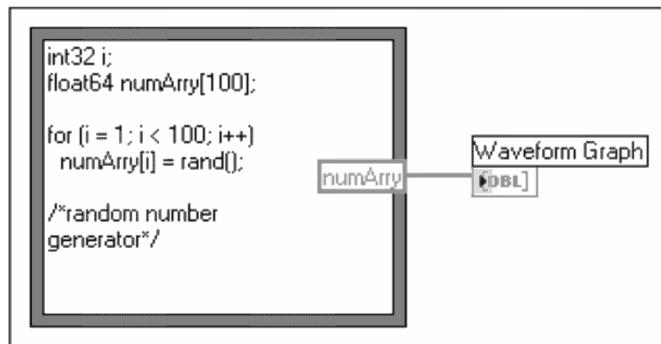


Figure 8.6 Formula Node.

When you work with variables, remember the following points:

- There is no limit to the number of variables or equations in a *Formula Node*.
- No two inputs and no two outputs can have the same name, but an output can have the same name as an input.
- Declare an input variable by right-clicking the *Formula Node* border and selecting *Add Input* from the shortcut menu. You cannot declare input variables inside the *Formula Node*.
- Declare an output variable by right-clicking the *Formula Node* border and selecting *Add Output* from the shortcut menu. The output variable name must match either an input variable name or the name of a variable you declare inside the *Formula Node*.
- You can change whether a variable is an input or an output by right-clicking it and selecting *Change to Input* or *Change to Output* from the shortcut menu.
- You can declare and use a variable inside the *Formula Node* without relating it to an input or output wire.

- You must wire all input terminals.
- Variables can be floating-point numeric scalars whose precision depends on the configuration of your computer. You also can use integers and arrays of numeric values for variables.
- Variables cannot have units.

Complete the following steps to change a *Formula Node* terminal from an output to an input or vice versa.

**Step 1:** Right-click the output or input terminal.

**Step 2:** Select *Change to Input* or *Change to Output* from the shortcut menu.

Complete the following steps to remove a terminal from a *Formula Node*.

**Step 1:** Right-click the input or output terminal you want to remove.

**Step 2:** Select *Remove* from the shortcut menu.

You also can select the terminal and press the <Delete> or <Backspace> key.

Input arrays and input-output arrays take their type from the array to which they are wired. These arrays do not require you to declare them inside the *Formula Node*. However, you must declare local arrays and output arrays in the *Formula Node*. Arrays are zero-based, as they are in C. Unlike C, LabVIEW treats an assignment to an array element that is out of range as a non-operation, and no assignment occurs. Also unlike C, if you make a reference to an array element that is out of range, LabVIEW returns a value of zero. You must declare array outputs in the *Formula Node* unless they correspond to an array input, in which case the two terminals must share a name.

### 8.6.2 Creating Formula Nodes

Complete the following steps to create a *Formula Node*.

**Step 1:** Place a *Formula Node* on the block diagram.

**Step 2:** Review the available functions and operators you can use.

**Step 3:** Use the labeling tool or the operating tool to enter the equations you want to calculate inside the *Formula Node*. Each assignment must have only a single variable on the left side of the assignment (=). Each assignment must end with a semicolon (;). Confirm that you are using the correct *Formula Node* syntax.

**Step 4:** If a syntax error occurs, click the broken *Run* button to display the *Error list* window. LabVIEW marks the syntax error with a # symbol.

**Step 5:** Create an input terminal for each input variable by right-clicking the *Formula Node* border and selecting *Add Input* from the shortcut menu. Type the variable name in the terminal that appears. You can edit the variable name at any time using the labeling tool or the Operating tool, except when the VI is running.

**Step 6:** Variable terminals are case sensitive. There is no limit to the number of terminals or equations in a *Formula Node*. You can change a terminal type or remove a terminal.

**Step 7:** Create an output terminal for each output variable by right-clicking the *Formula Node* border and selecting *Add Output* from the shortcut menu. Type the variable name in the terminal that appears. You can edit the variable name at any time using the labeling tool or the operating tool, except when the VI is running. Output variables have thicker borders than input variables.

**Step 8:** (Optional) The default data type for output terminals is double-precision, floating-point. To change the data type, create an input terminal with exactly the same name as the output terminal and wire a data type to that input terminal. Doing so also provides a default value for the terminal. You also can use the *Formula Node* syntax to define the variable inside the *Formula Node*. For example, int32 y; changes the data type of the output terminal y to 32-bit integer.

**Step 9:** Wire the input and output terminals of the *Formula Node* to their corresponding terminals on the block diagram. All input terminals must be wired. Output terminals do not have to be wired.

### 8.6.3 Formula Node Syntax

The *Formula Node* syntax is similar to the syntax used in text-based programming languages. Remember to end assignments with a semicolon (;) as in C. Use scope rules to declare variables in *Formula Nodes*. The *Formula Node* syntax is summarized below using Backus-Naur Form (BNF notation). The summary includes non-terminal symbols: compound-statement, identifier, conditional-expression, number, array-size, floating-point-type, integer-type, left-hand-side, assignment-operator, and function. Symbols in red bold monospace are terminal symbols given exactly as they should be reproduced. The symbol # denotes any number of the term following it.

Use the break keyword to exit the nearest Do, For, or While Loop or Switch statement in the *Formula Node*. Use the continue keyword to transfer control to the next iteration of the nearest Do, For, or While Loop in the *Formula Node*. The conditional statement uses the same syntax as the if and else statements in C. The do statement, the For statement, the switch statement and the while statement all use the same syntax as the respective statements in C.

### 8.6.4 Scope Rules for Declaring Variables in Formula Nodes

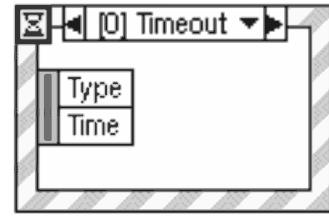
*Formula Nodes* use the same scope rules for declaring variables as C.

- All variables you declare in bracketed blocks are only accessible within the bracketed block.
- All input terminals are considered variables at the outermost block (not enclosed in brackets), and cannot be declared again in the outermost block.
- LabVIEW tries to match variables you declare at the outermost block (not enclosed in brackets) to output terminals with the same name.
- You can declare output terminals that have no corresponding input terminal and are not declared at the outermost block.

Use the correct *Formula Node* syntax when declaring variables.

## 8.7 EVENT STRUCTURE

The *Event* structure waits until an event happens and then executes the appropriate case to handle that event. Right-click the structure border to add new event cases and configure which events to handle. Wire a value to the *Timeout* terminal as shown at the top left of the *Event* structure to specify the number of milliseconds the *Event* structure should wait for an event to occur. The default is  $-1$ , indicating never to time out.



The *Event Data Node* is attached to the inside left and right borders of each event case. The node identifies the data LabVIEW returns when an event occurs. The node displays data that is different in each case of the *Event* structure depending on which event(s) you configure that case to handle. If you configure a single case to handle multiple events, only the data that is common to all handled event types is available. When you place an *Event* structure on the block diagram, the *Timeout* event case is the default case. When you place an *Event* structure on the block diagram from the *Application Control»Events* palette, the structure displays the *Event Dynamic Registration* terminals. When you place an *Event* structure on the block diagram from the *Structures* palette, the structure does not display the terminals. To display the *Event Dynamic Registration* terminals, right-click the *Event* structure and select *Show Dynamic Event Terminals* from the shortcut menu.

Complete the following steps to add or duplicate an event case.

**Step 1:** Right-click the *Event* structure border.

**Step 2:** Select *Add Event Case* or *Duplicate Event Case* from the shortcut menu. The *Edit Events* dialog box appears.

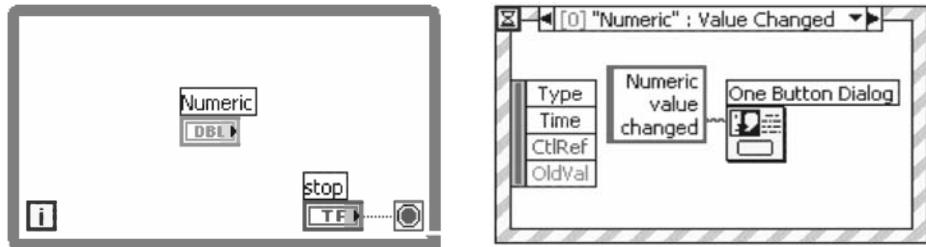
**Step 3:** Edit the event case.

When you duplicate an event case, LabVIEW does not duplicate the events you configured for that case. You must configure the new case to handle an event. If you configure multiple cases to handle the same event, the VI appears broken.

LabVIEW can generate events even when no *Event* structure is waiting to handle them. Because the *Event* structure handles only one event each time it executes, place the *Event* structure in a While Loop that terminates when the VI is no longer interested in events to ensure that an *Event* structure handles all events that occur. If no *Event* structure executes to handle an event and front panel locking is enabled, the user interface of the VI becomes unresponsive. If this occurs, click the *Abort* button to stop the VI. You can disable panel locking by right-clicking the *Event* structure and removing the checkmark from the **Lock front panel until the event case for this event completes** checkbox in the *Edit Events* dialog box. You cannot turn off front panel locking for filter events. For example, in the block diagram in Figure 8.7, the *Event* structure is outside the While Loop and front panel locking is enabled for the numeric *Value Change* case.

If you change the value of the numeric control, an event occurs. The *Event* structure executes once and handles the *Value Change* event. If you change the value of the numeric control again, another event occurs, and the user interface locks because front panel locking is enabled. The *Event* structure already executed once, and because it is not inside a While Loop, it is unable execute again to handle the second event. If you click the *stop* Boolean control to stop the While Loop and the VI,

the VI cannot stop because the block diagram still has not handled the second event, and LabVIEW does not process the event when you click the *stop* Boolean control until the block diagram handles that event. You can avoid this behavior by placing the *Event* structure inside the While Loop.



**Figure 8.7** Event structure outside a loop.

*Event* structures begin waiting for events as soon as the VI runs. For example, if you configure two *Event* structures within the same *Sequence* structure to wait for a mouse click on the same front panel object, both *Event* structures execute as soon as the event occurs for the first time. If you want the second *Event* structure to execute the second time the event occurs, do not use a *Sequence* structure. Instead, place one *Event* structure inside a For Loop that you configure to execute twice.

## 8.8 LabVIEW MathScript

LabVIEW MathScript is a text-based language you can use to write functions and scripts for use in the LabVIEW MathScript Window or MathScript Node. The MathScript syntax is similar to the MATLAB language syntax. In LabVIEW MathScript, you generally can execute scripts written in the MATLAB language syntax. However, the MathScript engine executes the scripts, and the MathScript engine does not support some functions that the MATLAB software supports.

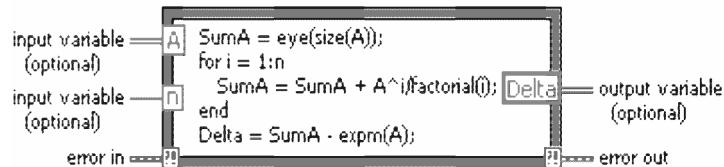
### 8.8.1 LabVIEW MathScript Window

Select *Tools»MathScript Window* to display this window. Use this window to edit and execute mathematical commands, create mathematical scripts, and view numerical and graphical representations of variables. This window generates output from and maintains a history of commands that you call, lists variables that you define, and displays variables that you select. In LabVIEW MathScript, you generally can execute scripts written in the MATLAB language syntax. However, the MathScript engine executes the scripts, and the MathScript engine does not support some functions that the MATLAB software supports. Scripts you run in the *LabVIEW MathScript Window* can generate errors.

You can save scripts you create in the *LabVIEW MathScript Window* or load scripts from file into the *LabVIEW MathScript Window*. You also can save and load data files. The MathScript Node and the MathScript Window can communicate only if they are in the same application instance. Use the application instance name that appears in the bottom-left corner of the *LabVIEW MathScript Window* to identify which application instance a *LabVIEW MathScript Window* belongs to. The application instance name includes the project name followed by the target name.

### 8.8.2 MathScript Node

The MathScript Node executes LabVIEW MathScripts. You can use it to evaluate scripts that you create in the LabVIEW MathScript Window. The LabVIEW MathScript syntax is similar to the MATLAB language syntax as shown below.



Use this node to execute scripts. Enter the script in the node or right-click the node border to import text into the node. Right-click the node border to add input and output variables. Right-click an input or output variable to set its data type. When you create a LabVIEW MathScript, you must use supported data types. The MathScript Node and the MathScript Window can communicate only if they are in the same application instance.

### 8.8.3 Creating a LabVIEW MathScript

Complete the following steps to create and run a VI that uses a LabVIEW MathScript.

**Step 1:** Place a MathScript Node on the block diagram.

You can create MathScript Nodes only in the LabVIEW Full and Professional Development Systems. If a VI contains a MathScript Node, you can run the VI in all LabVIEW packages.

**Step 2:** Use the operating or labeling tool to enter the script in the MathScript Node:

```
a=rand(50, 1)
plot(a)
```

**Step 3:** Add an output to the MathScript Node and create an indicator for the output.

- (a) Right-click the MathScript Node frame and select *Add Output* from the shortcut menu. Enter *a* in the output terminal to add an output for the *a* variable in the MathScript. By default, the MathScript Node includes one input and one output for the *error in* and *error out* parameters.
- (b) Change the data type of the output terminal. In MathScript, the default data type for any new input or output is *a Scalar»DBL*. Right-click the *a* output and select *Choose Data Type»Matrix»Real Matrix* from the shortcut menu.
- (c) Right-click the output terminal and select *Create»Indicator* from the shortcut menu to create a matrix indicator labeled *Real Matrix*.

**Step 4:** Right-click the *error out* output terminal and select *Create»Indicator* from the shortcut menu to create an *error out* cluster indicator labeled *error out*.

**Step 5:** Run the VI. LabVIEW invokes the MathScript server, creates a vector of random values, plots that information to a graph, and displays the values that make up the vector in the *Real Matrix* indicator on the front panel.

#### 8.8.4 Defining a MathScript Function or Script

You can define functions and create scripts to use in the LabVIEW MathScript Window or in the MathScript Node. Complete the following steps to create a function or script and prepare it for use in the *LabVIEW MathScript Window* or MathScript Node. If you define a function with the same name as a built-in LabVIEW MathScript function, LabVIEW executes the function you defined instead of the original MathScript function. When you execute the help command, LabVIEW returns help content only for the function you defined. You cannot retrieve the help content for the original MathScript function.

**Step 1:** In the *Script Editor* on the *Script* page of the *LabVIEW MathScript Window*, create a function or script using the proper syntax. You also can use a text editor to create the function or script.

**Step 2:** Save the function or script. The filename for a function must be the same as the name of the function and must have a lowercase .m extension. For example, the filename for the *foo* function must be *foo.m*. Use unique names for all functions and scripts. You must save the function or script in a directory that you specified in the *Path* section of the MathScript Preferences dialog box.

**Step 3:** (Optional) Select *File»Save & Compile Script* to compile the script you created.

#### 8.8.5 Debugging Scripts

Use the following programming techniques to make debugging the script easier:

- Write the script and run it in the native program for testing and debugging purposes before you import it into LabVIEW.
- Verify data types. When you create a new input or output, make sure that the data type of the terminal is correct. Also, create controls and indicators for inputs and outputs so that you can monitor what values are being passed between LabVIEW and the script server engine. This allows you to pinpoint where a script node calculates a value incorrectly, if necessary.
- Take advantage of the error-checking parameters for debugging information. Create an indicator for the *error out* terminal on a MathScript Node, a MATLAB script node, or an Xmath Script Node before you run any VI so you can view the generated error information at run time.

#### 8.8.6 Clearing, Saving and Loading Scripts

Complete the following steps to clear the contents of a script node.

**Step 1:** Right-click a script node.

**Step 2:** Select *Clear Script* from the shortcut menu.

You can save scripts that you create in the *Script Editor* of the LabVIEW MathScript Window. Complete the following steps to save a script.

**Step 1:** In the *LabVIEW MathScript Window*, select *File»Save Script*. You also can click the *Save* button on the *Script* page of the *LabVIEW MathScript Window*.

**Step 2:** In the File dialog box, navigate to the directory in which you want to save the script.

**Step 3:** Enter a name for the script in the *File name* field. The name must have a lowercase .m extension if you want LabVIEW to run the script.

**Step 4:** Click the *OK* button to save the script.

You can compile a script when you save it to decrease the run-time compilation time. In the *LabVIEW MathScript Window*, select *File»Save & Compile Script*. You can load a saved script in the MathScript Node or paste scripts directly from the *Script Editor* into the MathScript Node and vice versa. You also can load existing scripts into the *LabVIEW MathScript Window*. In the *LabVIEW MathScript Window*, select *File»Load Script* or click the *Load* button on the *Script* page to load the script you want.

You can save and load data files in the LabVIEW MathScript Window. Data files contain numerical values for variables. Complete the following steps to save a data file.

**Step 1:** In the *LabVIEW MathScript Window*, select *File»Save Data*. You also can right-click the *Variable List* on the *Variables* page and select *Save Data* from the shortcut menu.

**Step 2:** In the File dialog box, navigate to the directory in which you want to save the data file.

**Step 3:** Enter a name for the data file in the *File name* field.

**Step 4:** Click the *OK* button to save the data file.

You also can load existing data files into the *LabVIEW MathScript Window*. In the *LabVIEW MathScript Window*, select *File»Load Data* or right-click the *Variable List* on the *Variables* page and select *Load Data* from the shortcut menu to load the data file you want. You must save data files before you can load them into the *LabVIEW MathScript Window*.

### 8.8.7 Importing or Exporting Scripts

If you already have a script written in the LabVIEW MathScript syntax, the MATLAB language syntax, or the Xmath syntax, you can import it into LabVIEW. You also can save a script written in LabVIEW to a text file.

Complete the following steps to import a script into a script node in LabVIEW.

**Step 1:** Select a MathScript Node, a MATLAB script node, or an Xmath Script Node and drag out a region to place the script node on the block diagram.

- MathScript Node
- MATLAB script node
- Xmath Script Node

You can create script nodes only in the LabVIEW Full and Professional Development Systems. If a VI contains a script node, you can run the VI in all LabVIEW packages.

**Step 2:** Right-click the script node and select *Import* from the shortcut menu to display a file dialog box.

**Step 3:** Select the file you want to import and click the *OK* button. The script text appears in the node.

Complete the following steps to save a script to a text file.

**Step 1:** Right-click the MathScript Node, MATLAB script node, or Xmath Script Node and select *Export* from the shortcut menu to display a file dialog box.

**Step 2:** Enter the name to use for the text file or select a file to overwrite.

**Step 3:** Click the *OK* button.

**Step 4:** Open the script in the LabVIEW MathScript Window, the MATLAB software or the Xmath software.

MathScripts, scripts written in the MATLAB language syntax, and Xmath scripts are text files. Although text files usually have a .txt extension, MathScript, like the MATLAB software and other mathematics software, processes scripts with a .m extension, and the Xmath software uses a .ms extension.

### 8.8.8 Configuring the Data Type of Script Node Terminals

The LabVIEW MathScript syntax, the MATLAB language syntax and the Xmath syntax are loosely-typed script languages and do not determine the data type of a variable until after the script runs. Therefore, LabVIEW cannot determine the type of a variable in edit mode. LabVIEW queries the script server to find out possible data types. You can select the LabVIEW data type for each terminal.

If you do not correctly configure a variable data type, LabVIEW might produce an error, return incorrect information at run time, or both.

Complete the following steps to change the data type of an input or output terminal on a MathScript Node, a MATLAB script node, or an Xmath Script Node.

**Step 1:** Right-click the terminal of the input or output and select *Choose Data Type* from the shortcut menu. A list of the available data types appears.

**Step 2:** Select the data type you want to use.

LabVIEW recognizes many of the data types available in MathScript, the MATLAB software, and the Xmath software, although the data types might be named differently.

### 8.8.9 MathScript Function Syntax

You can define functions to use in the LabVIEW MathScript Window or in the MathScript Node. If you define a function with the same name as a built-in LabVIEW MathScript function, LabVIEW executes the function you defined instead of the original MathScript function. When you execute the help command, LabVIEW returns help content only for the function you defined. You cannot retrieve the help content for the original MathScript function.

A LabVIEW MathScript function must use the following syntax:

```
function outputs = function_name(inputs)
% documentation
script
```

Begin each function definition with a function name *outputs* to list the output variables of the function. If the function has more than one output variable, enclose the variables in square brackets and separate the variables with white space or commas. *function\_name* is the name of the function you want to define. *inputs* lists the input variables to the function. Use commas to separate the input variables. *documentation* is the help content that you want LabVIEW to return for the function when you execute the help command. Precede each line of help documentation with a % character. You can place help content anywhere in the function definition. However, LabVIEW returns only the first comment block in the *Output Window*. You can use all other comment blocks for internal documentation. *script* defines the executable body of the function.

If you define multiple functions in one MathScript file, all functions following the first are subfunctions and are accessible only to the main function. A function can call only those functions that you define below it. You cannot call functions recursively. After you define a function, save the function. The filename for a function must be the same as the name of the function and must have a lowercase .m extension.

### 8.8.10 MathScript Syntax

Use LabVIEW MathScript to write functions and scripts for use in the LabVIEW MathScript Window or MathScript Node. The MathScript syntax is similar to the MATLAB language syntax. Use the following guidelines when writing MathScript functions and scripts:

- You cannot define variables that begin with an underscore, white space or a digit.
- MathScript variables adapt to data types. For example, if *a* =  $\sin(3\pi/2)$ , then *a* is a double-precision floating-point number. If *a* = ‘result’, then *a* is a string.
- You can use either *i* or *j* to represent the imaginary unit equal to the square root of  $-1$ .
- Use white space or commas to separate matrix elements, and use semicolons to separate rows of a matrix.
- You cannot use *end* to specify the last element of a series or matrix.
- You cannot use *n*-dimensional arrays.
- You cannot use cell arrays.
- If you end a command line with a semicolon, the *LabVIEW MathScript Window* does not display the output for that command. Some functions, such as *disp*, display output even if you end the command line with a semicolon.
- In localized versions of LabVIEW, you cannot use a comma as a decimal separator. You must use a period as a decimal separator.

## SUMMARY

- The *Select* function selects between two inputs depending on a third Boolean input.
- A *Case* structure has two or more subdiagrams, or cases. Only one subdiagram is visible at a time, and the structure executes only one case at a time.
- If the case selector terminal is a Boolean value, the structure has a TRUE case and a FALSE case. If the selector terminal is an integer, string, or enumerated type value, the structure can have up to  $2^{31} - 1$  cases.

- Inputs are available to all subdiagrams of a *Case* structure, but subdiagrams do not need to use each input. If output tunnel is not defined, in all cases it appears as white square.
- When creating a subVI from a *Case* structure, wire the error input to the selector terminal, and place all subVI codes within the *No Error* case to prevent the subVI from executing if it receives an error.
- Timed Loop—Executes a subdiagram until a condition is met or interminably.
- Timed Sequence—Executes multiple subdiagrams in sequence.
- Timed Loop with Frames—Executes multiple subdiagrams in sequence until a condition is met or interminably. Add frames to a Timed Loop to create a Timed Loop with frames.
- *Formula Nodes* are useful for equations that have many variables. Each equation statement must terminate with a semicolon ( ; ).
- The *Event* structure waits until an event happens, and then executes the appropriate case to handle that event.
- The LabVIEW MathScript syntax is similar to the MATLAB language syntax.

### MISCELLANEOUS SOLVED PROBLEMS

**Problem 8.1** Create a VI to add or subtract two numbers. Use *Case* structures to switch between addition and subtraction.

**Solution** The front panel and the block diagram to solve the problem using the Boolean, menu and text ring to switch between add and subtract are shown in Figures P8.1(a) and P8.1(b).

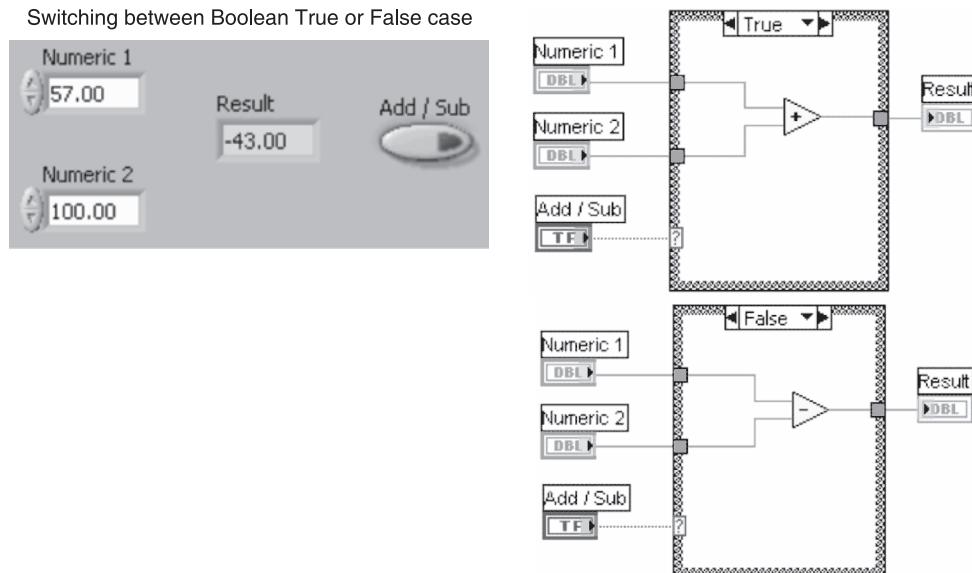
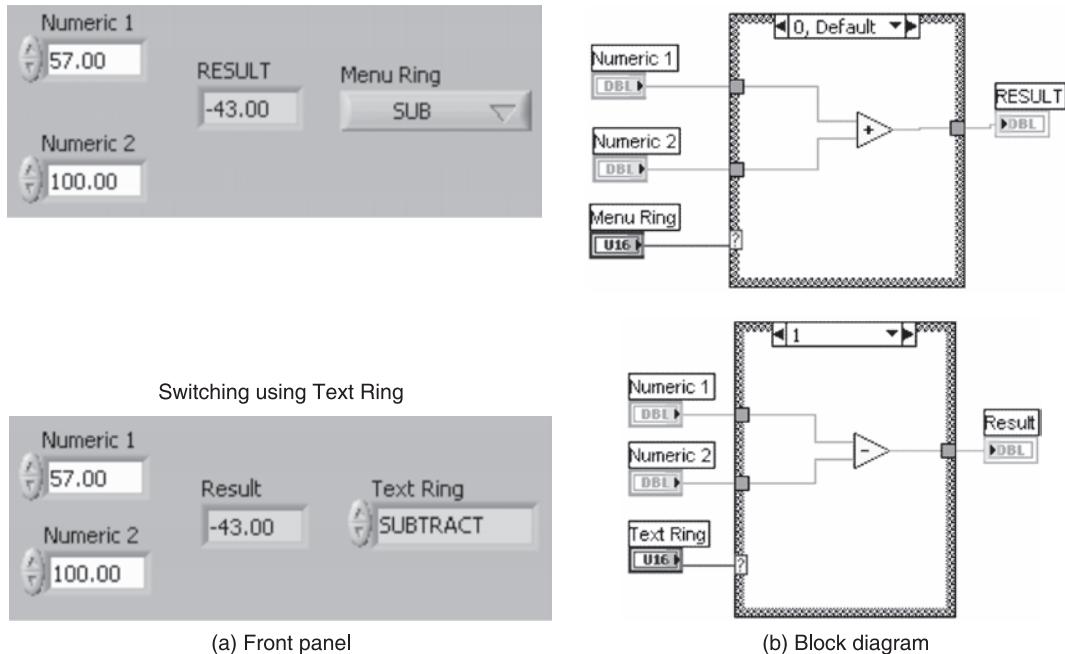


Figure P8.1 (Contd.)

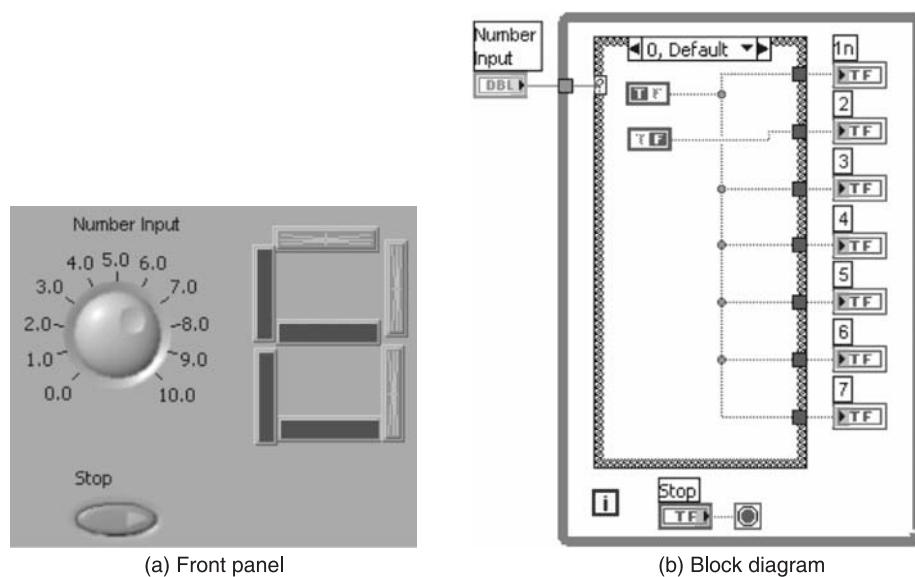
## Switching using Menu Ring



**Figure P8.1**

**Problem 8.2** Build a VI to create a seven-segment LED display.

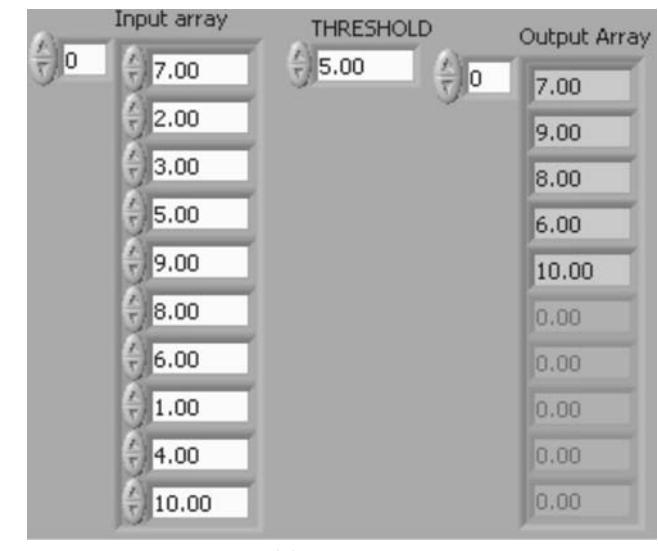
**Solution** To build a seven-segment LED create the front panel and the block diagram as shown in Figures P8.2(a) and P8.2(b).



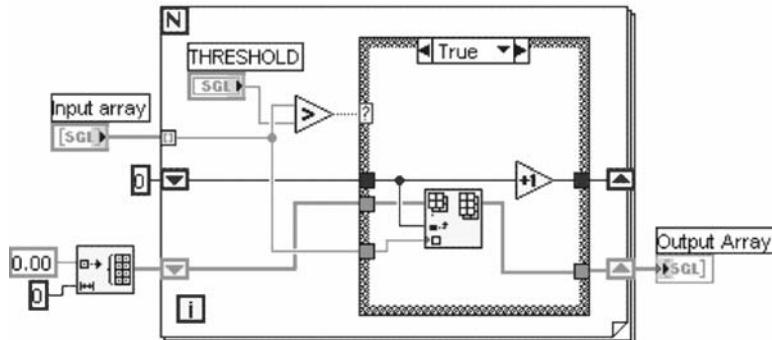
### Figure P8.2

**Problem 8.3** Build a VI which consists of numeric input array. Set a threshold value and separate the array elements which are greater than the threshold. Create an icon and connector and save this VI as subVI.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P8.3(a) and P8.3(b).



(a) Front panel



(b) Block diagram

**Figure P8.3**

**Problem 8.4** Create a VI to generate  $n$  number of random numbers. Set the threshold and separate the numbers which are greater than the threshold. Build this VI using the SubVI created in Problem 8.3.

**Solution** The front panel and the block diagram shown in Figures P8.4(a) and P8.4(b) should be built to create a VI to generate  $n$  number of random numbers.

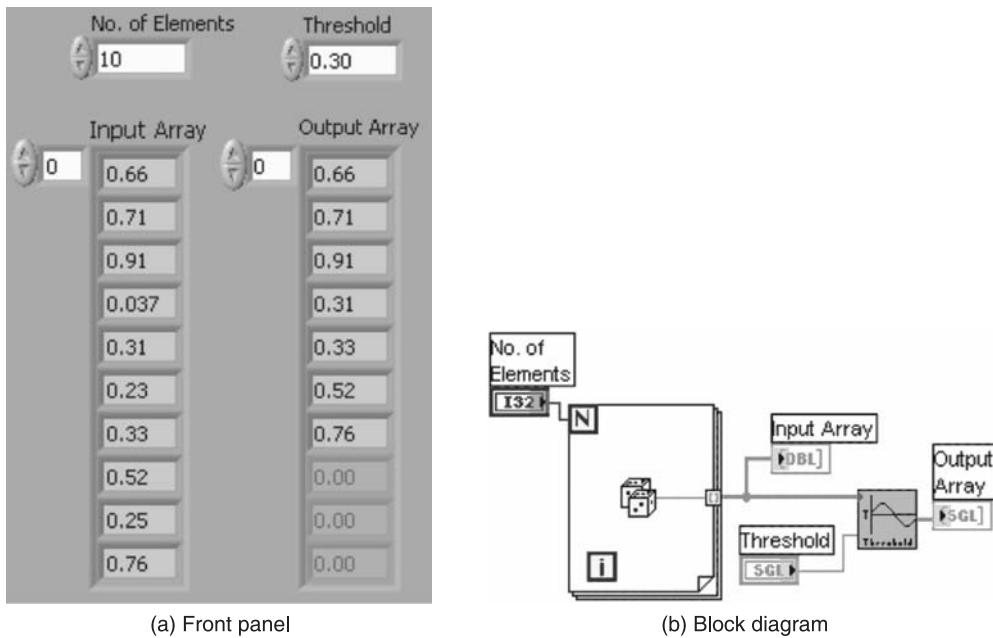
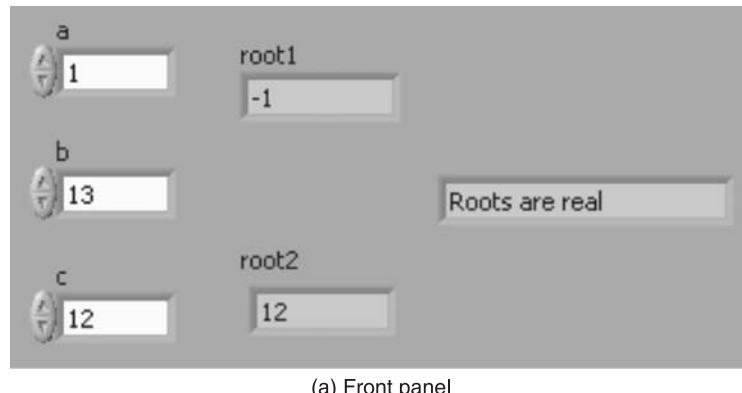


Figure P8.4

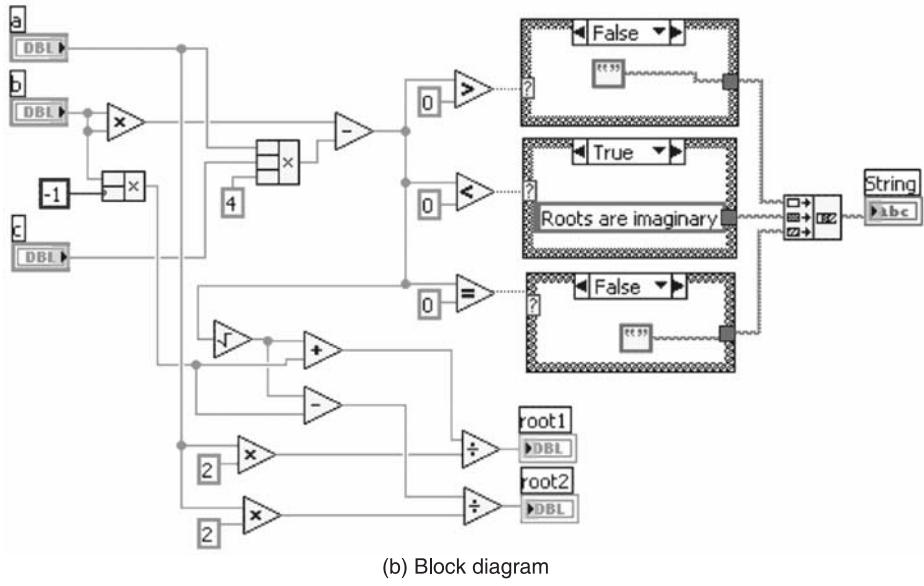
**Problem 8.5** Build a VI to find the roots of a quadratic equation. Input the coefficients of  $x^2$ ,  $x$  and constant as  $a$ ,  $b$  and  $c$  respectively. Display the roots and the message if the roots are real or imaginary.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P8.5(a) and P8.5(b).



(a) Front panel

Figure P8.5 (Contd.)

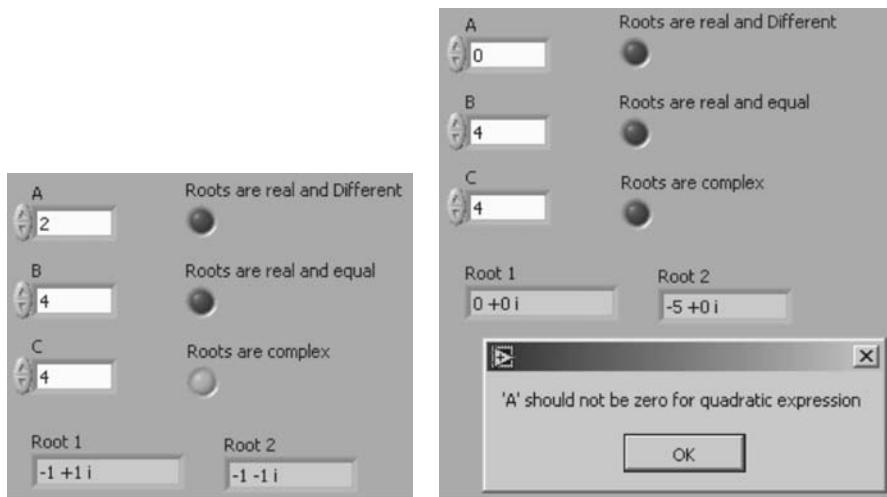


(b) Block diagram

**Figure P8.5**

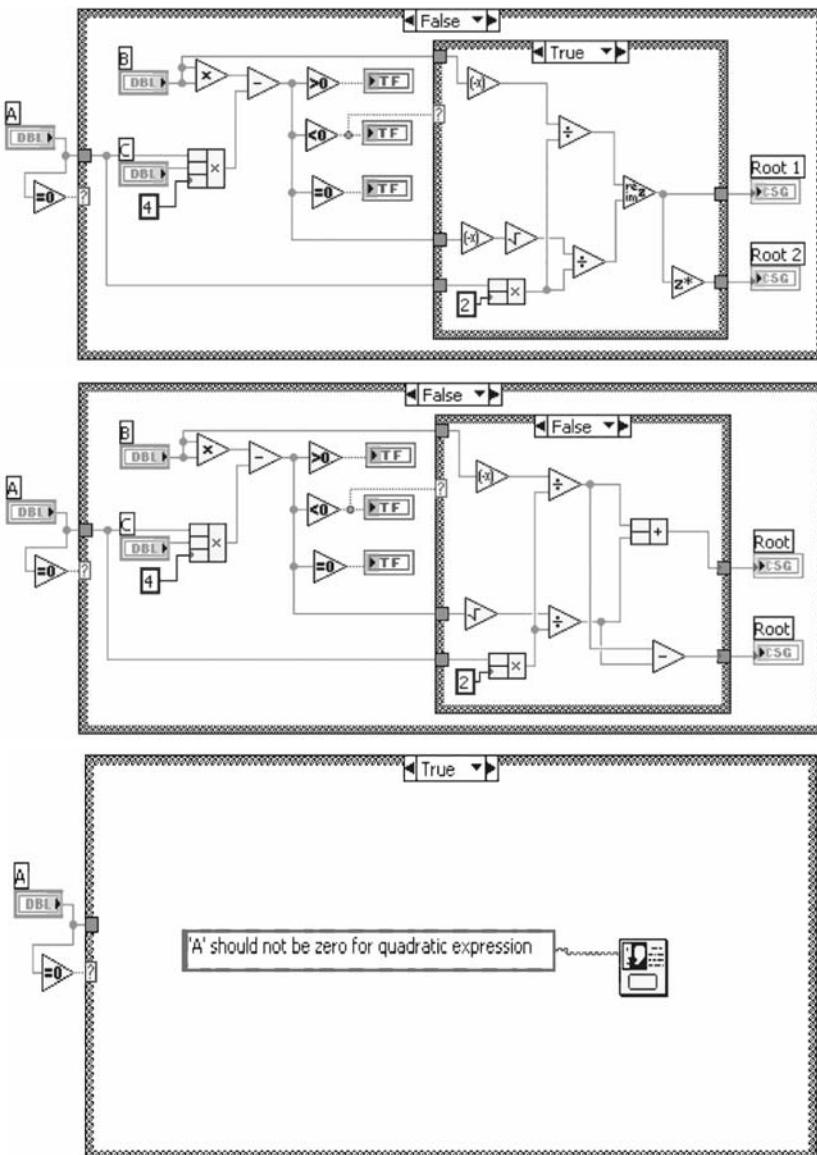
**Problem 8.6** Build a VI to find the roots of a quadratic equation with complex roots. Input the coefficients of  $x^2$ ,  $x$  and constant as  $A$ ,  $B$  and  $C$  respectively. The LED should glow when roots are real and different or roots are real and equal or when the roots are complex. Display the message ‘ $A$ ’ should not be zero for quadratic expression” is the coefficient of  $x^2$  is zero.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P8.6(a) and P8.6(b).



(a) Front panel

**Figure P8.6 (Contd.)**

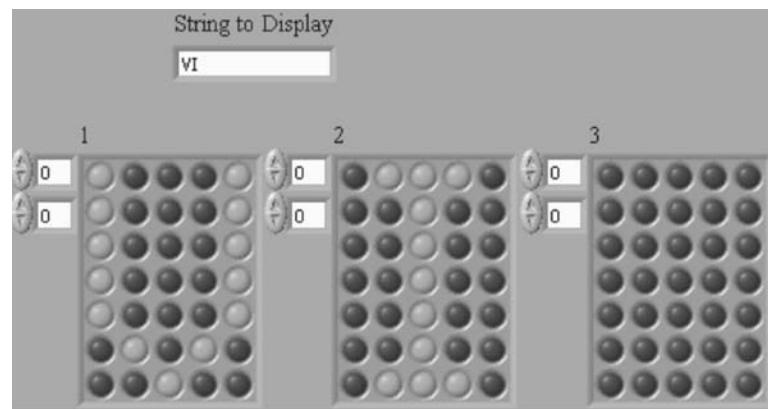


(b) Block diagram

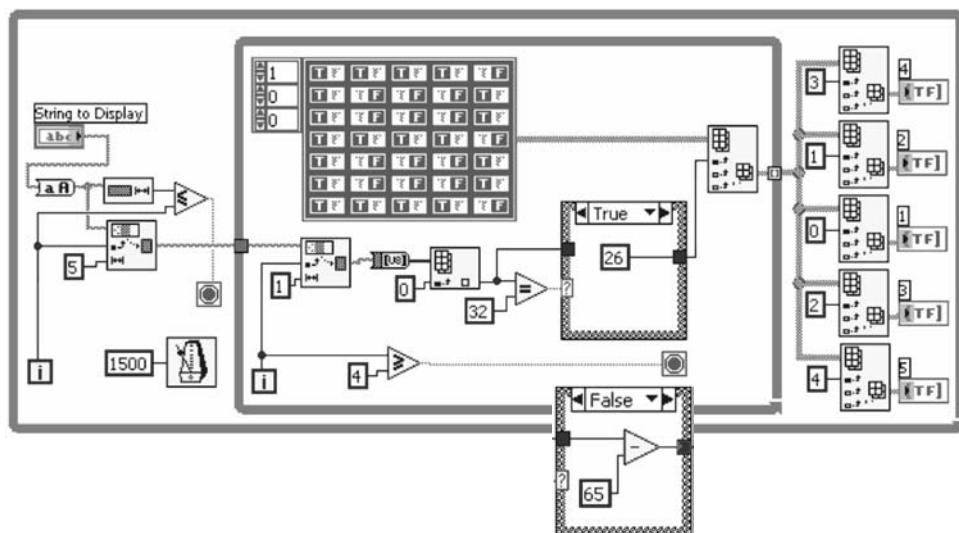
**Figure P8.6**

**Problem 8.7** Build a VI to display a string in the array of LEDs. Each letter has to be displayed in a separate array and the letters have to move from one array to another in the direction of left to right.

**Solution** Create the front panel and the block diagram to solve the problem as shown in Figures P8.7(a) and P8.7(b).



(a) Front panel

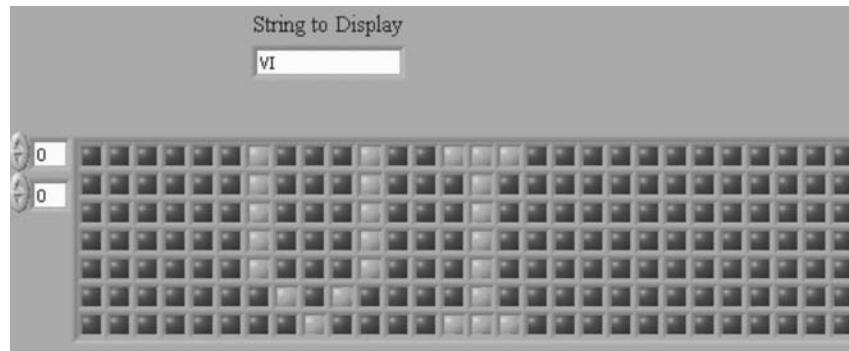


(b) Block diagram

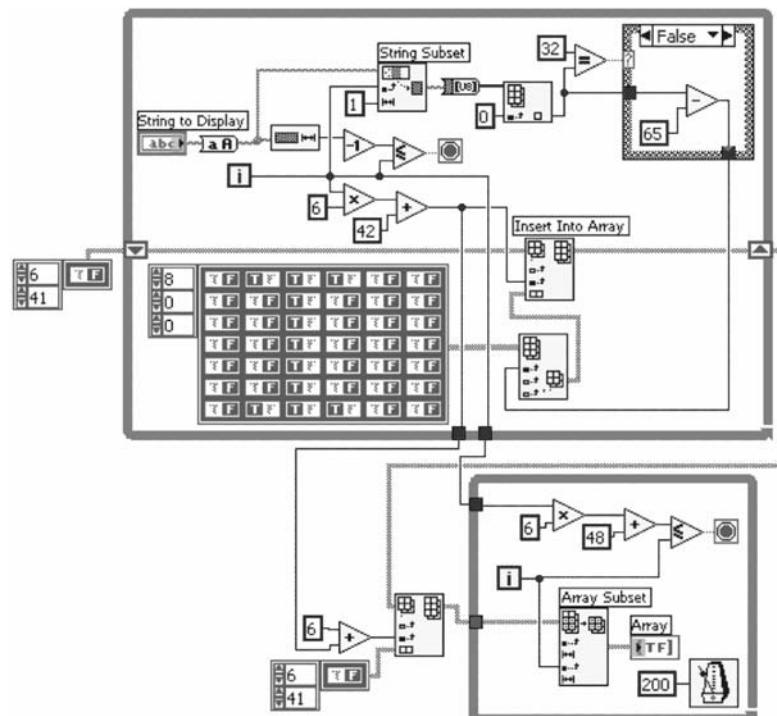
**Figure P8.7**

**Problem 8.8** Instead of a separate array display as in the previous problem, the string in this problem is a single array and the letters move from left to right.

**Solution** To display moving letters, create the front panel and the block diagram as shown in Figures P8.8(a) and P8.8(b).



(a) Front panel



(b) Block diagram

**Figure P8.8**

**Problem 8.9** Build a VI to compute the following equations using formula nodes.

$$y_1 = x^3 + x^2 + 5;$$

$$y_2 = m*x + b;$$

**Solution** The front panel and the block diagram to solve the equation using the formula node are shown in Figures P8.9(a) and P8.9(b).

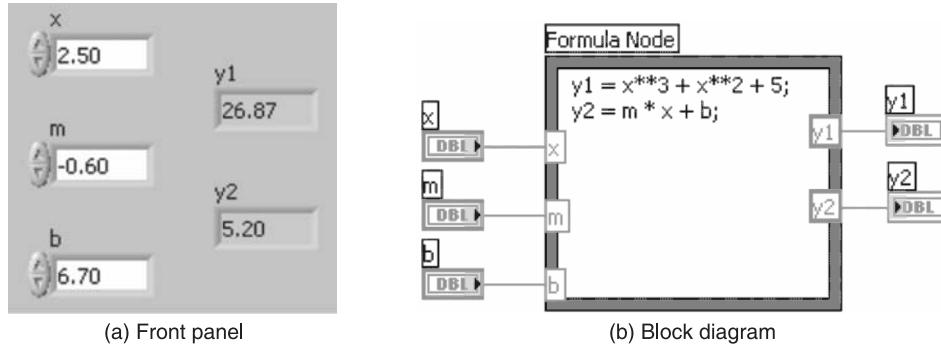


Figure P8.9

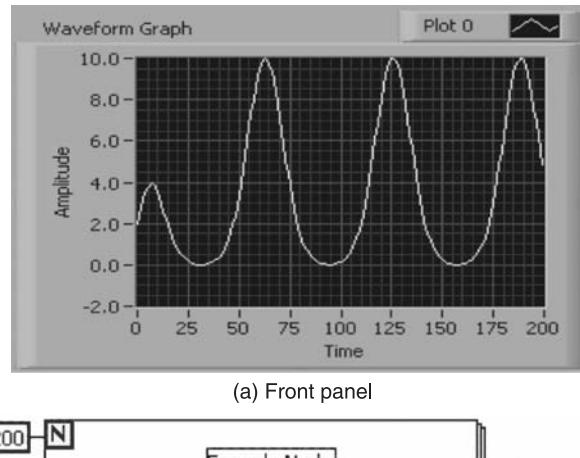
**Problem 8.10** Build a VI to execute the following equation to get the waveform as shown in the figure.

$$a = \tanh(x) + \cos(x)$$

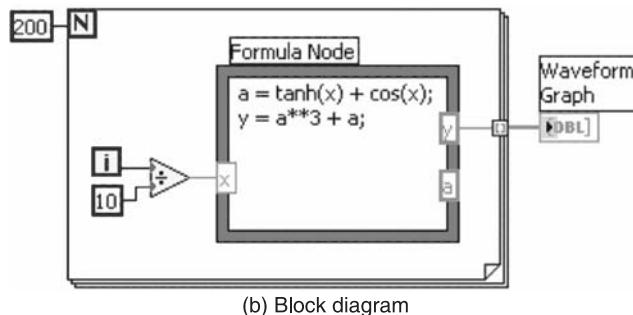
$$y = a^3 + a$$

where  $x$  varies from 0 to 20 in steps of 0.1.

**Solution** The front panel and the block diagram to solve the equation using the formula node and to plot on a waveform graph are shown in Figures P8.10(a) and P8.10(b).



(a) Front panel



(b) Block diagram

Figure P8.10

**Problem 8.11** Build a VI to execute the following expression using stacked sequence structure.

$$(A+B)/[(A+B)*2]$$

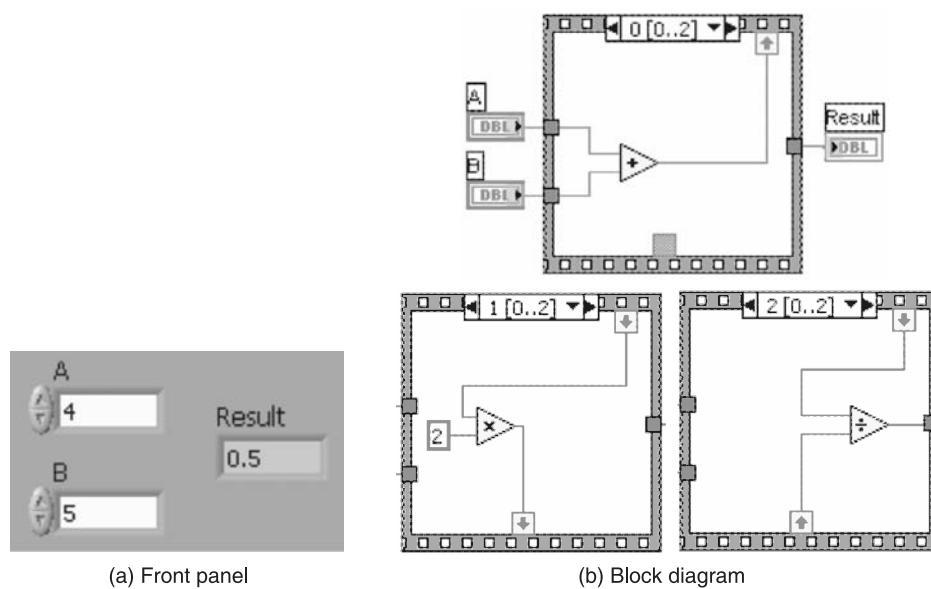
The three cases be:

Case1:  $A+B$

Case2:  $(A+B)/2$

Case3:  $(A+B)/[(A+B)*2]$

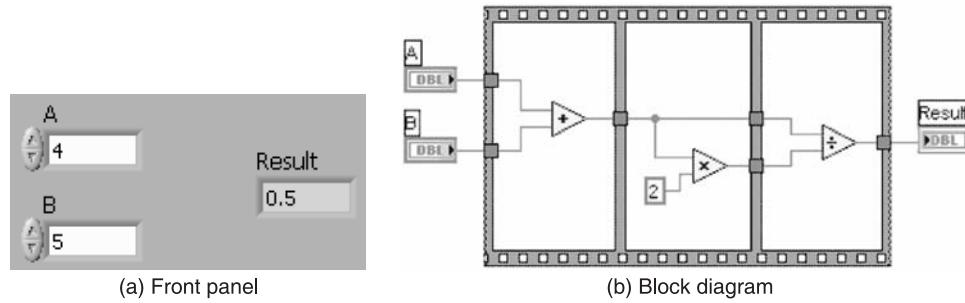
**Solution** To execute the expression using a stacked sequence structure, create the front panel and the block diagram as shown in Figures P8.11(a) and P8.11(b).



**Figure P8.11**

**Problem 8.12** Build a VI to execute the expression shown in the above exercise using the flat sequence structure. The three cases are the same as in the previous exercise.

**Solution** The front panel and the block diagram to solve the problem using the flat sequence structure are shown in Figures P8.12(a) and P8.12(b).



**Figure P8.12**

**Problem 8.13** Build a VI to generate random numbers. Multiply the random numbers by 10. Input any numbers which has to be matched with random numbers. Run the VI and find the numbers of iterations taken for matching. Also find the time taken for matching.

**Solution** The front panel and the block diagram to build a VI to generate random numbers are shown in Figures P8.13(a) and P8.13(b).

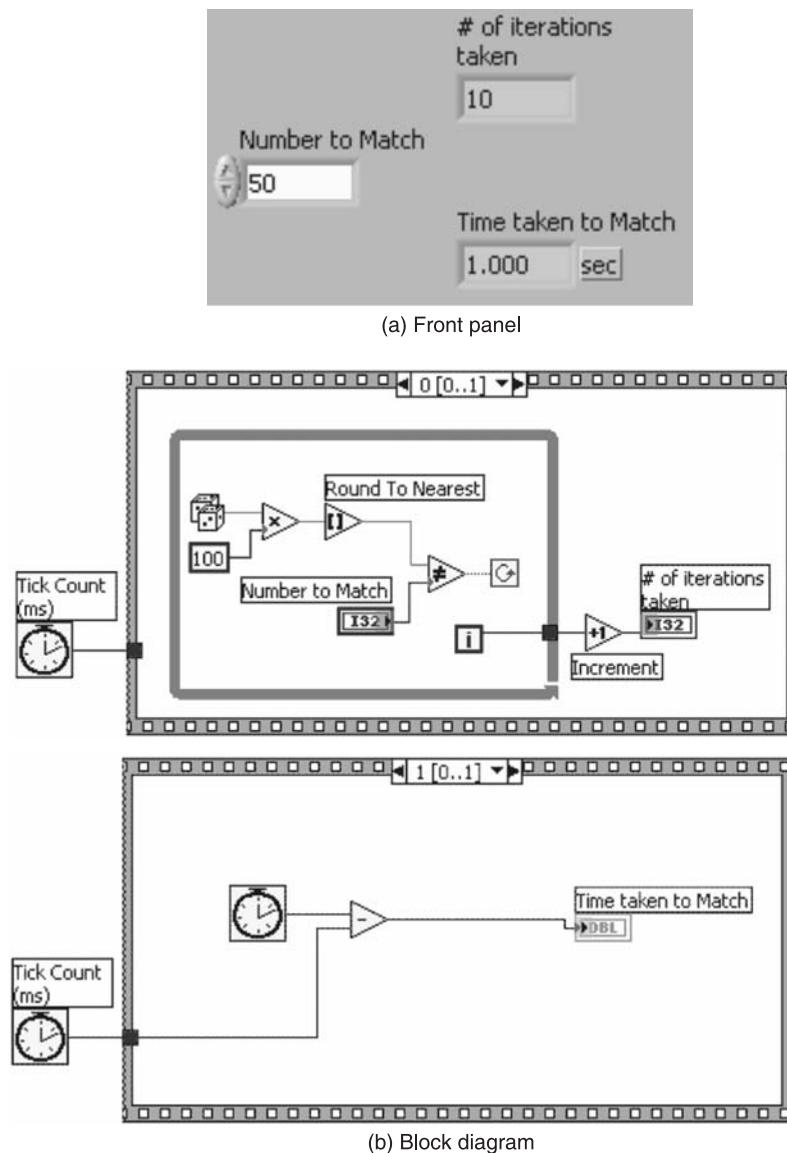


Figure P8.13

---

## REVIEW QUESTIONS

---

1. List the type of structures available in LabVIEW.
2. How can a new case be added in the case structure?
3. What is the purpose of a case selector in the case structure? What happens to the case structure if a numeric control is connected to the case selector?
4. Where are flat and stacked sequence structures used?
5. Enumerate the usage of timed structures.
6. How can a timed structure be used with data acquisition?
7. When are event structures used? List the limitations in using event structures.
8. Explain the method of adding input and output variables in a formula node. What are the limitations in the number of variables used?
9. What are the two methods of editing and executing MATLAB scripts in LabVIEW?
10. What is the difference between a formula node and a MathScript node?
11. Explain the usage of
  - (a) MathScript Node
  - (b) Event structure
12. Explain the procedure of adding an event in the event structure?
13. Is it advisable to use two event structures inside a loop? Justify the answer.
14. Explain the working method of
  - (a) Diagram Disable structure
  - (b) Conditional Disable structure

---

## EXERCISES

---

1. Build a VI to find the square root of a given number. If the given number is a negative display the message “Error..... Negative Number”.
2. Create a VI which consists of a numeric array with even and odd elements. Separate the odd and even elements in two different arrays.
3. Build a VI using case structures to switch between addition, subtraction, multiplication and division of two numbers.
4. Build a VI using a formula node to find the square root of the given number.
5. Compute the following equations using the formula node.

$$y = [(a \times b) + (b \times c)]/d$$

$$z = [(a \times y) + (b \times y)]$$

6. Build a VI to compute the following equations and plot the results on a waveform graph.

$$y1 = x^3 + x^2 - 5$$

$$y2 = x^2 + 4$$

where  $x$  varies from 0 to 10 in steps of 0.2.



## STRINGS AND FILE I/O

### 9.1 INTRODUCTION

A string is a sequence of displayable or non-displayable ASCII characters. Strings provide a platform-independent format for information and data. Some of the more common applications of strings include the following:

- Creating simple text messages.
- Passing numeric data as character strings to instruments and then converting the strings to numeric values.
- Storing numeric data to disk. To store numeric data in an ASCII file, you must first convert numeric data to strings before writing the data to a disk file.
- Instructing or prompting the user with dialog boxes.

On the front panel, strings appear as tables, text entry boxes, and labels. LabVIEW includes built-in VIs and functions you can use to manipulate strings, including formatting strings, parsing strings, and other editing.

### 9.2 CREATING STRING CONTROLS AND INDICATORS

The string control and indicator is located on the *Controls»Text Controls* and *Controls»Text Indicators* palettes to simulate text entry boxes and labels. Right-click a string control or indicator on the front panel to select from the display types. Table 9.1 shows an example message for each display type.

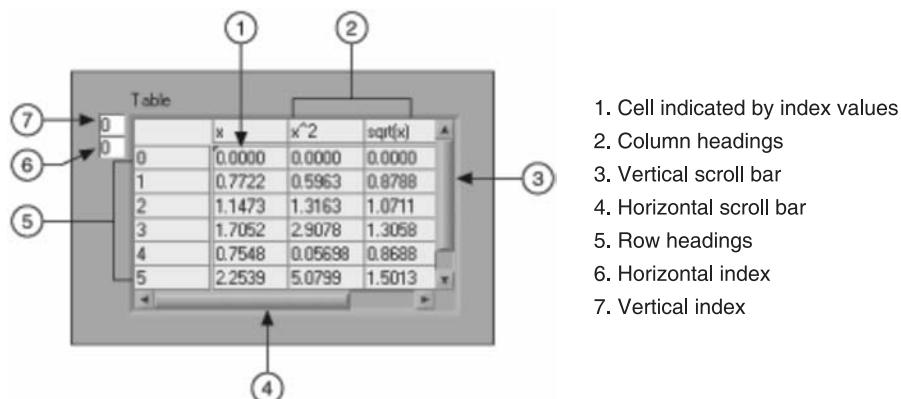
**TABLE 9.1** Example message for each display type

Display Type	Description	Message
Normal Display	Displays printable characters using the font of the control. Non-printable characters generally appear as boxes.	There are four display types. \\ is a backslash.
'\'' Codes Display	Displays backslash codes for all non-displayable characters.	There\\s are four display types. \\n\\w\\s\\a\\s\\b\\ackslash.
Password Display	Displays an asterisk (*) for each character including spaces.	***** *****
Hex Display	Displays the ASCII value of each character in hex instead of the character itself.	5468 6572 6520 6172 6520 666F 7572 2064 6973 706C 6179 2074 7970 6573 2E0A 5C20 6973 2061 2062 6163 6B73 6C61 7368 2E

Use the operating tool or labeling tool to type or edit text in a string control. Use the positioning tool to resize a front panel string object. To minimize the space that a string object occupies, right-click the object and select the *Visible Items»Scrollbar* option from the shortcut menu.

### 9.2.1 Tables

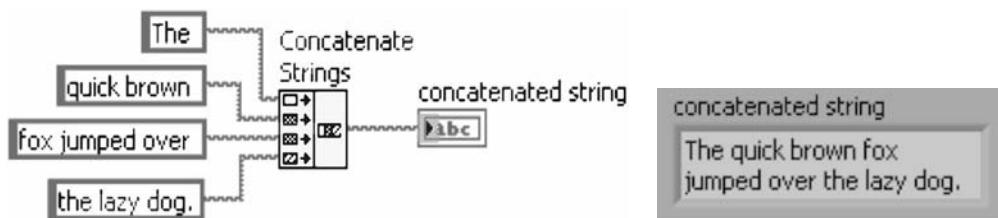
Use the table control to create a table on the front panel. Use the table control located on the *Controls»All Controls»List & Table* palette or the Express Table VI located on the *Controls»Text Indicators* palette to create a table on the front panel. Each cell in a table is a string, and each cell resides in a column and a row. A table is a display for a 2D array of strings. Figure 9.1 shows a table and all its parts. Define cells in the table by using the operating tool or the labeling tool to select a cell and typing text in the selected cell. The table displays a 2D array of strings, so you must convert 2D numeric arrays to 2D string arrays before you can display them in a table indicator. The row and column headers are not automatically displayed as in a spreadsheet. You must create 1D string arrays for the row and column headers.

**Figure 9.1** Table and all its parts.

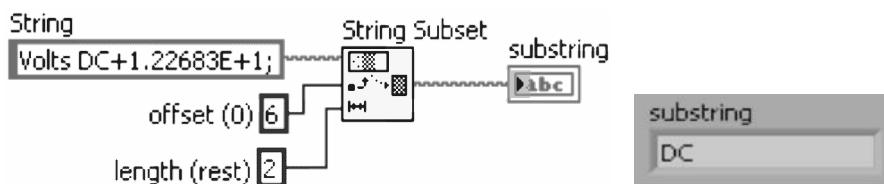
### 9.3 STRING FUNCTIONS

*String* functions behave similarly to array functions; in fact, strings are really just arrays of ASCII data. Use the *String* functions to concatenate two or more strings, extract a subset of strings from a string, convert data into strings, and format a string for use in a word processing or spreadsheet application. Use the *String* functions located on the *Functions»All Functions»String* palette to edit and manipulate strings on the block diagram. *String* functions include the following:

- **String Length**—Returns in length the number of characters (bytes) string, including space characters. For example, the *String Length* function returns a length of 19 for the following string: *The quick brown fox.*
- **Concatenate String**—Concatenates input strings and 1D arrays of strings into a single output string. For array inputs, this function concatenates each element of the array. Add inputs to the function by right-clicking an input and selecting *Add Input* from the shortcut menu or by resizing the function.

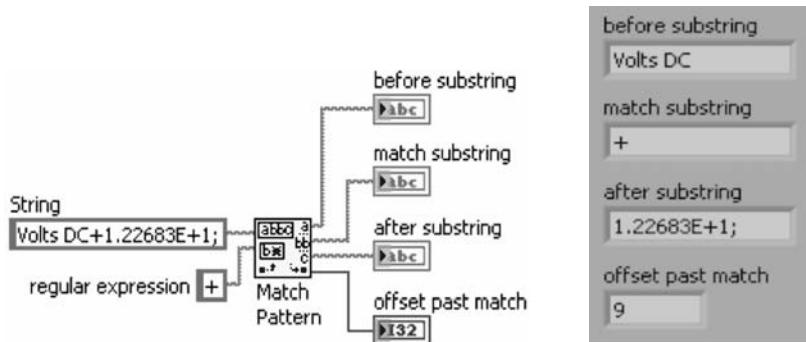


- **String Subset**—Returns the substring of the input string beginning at offset and containing length number of characters. The offset of the first character in string is 0. For example, if you use the string Volts DC+1.22683E+1; as the input, the *String Subset* function returns the following substring for an offset of 6 and a length of 2: DC.



- **Match Pattern**—Searches for regular expression in string beginning at offset, and if it finds a match, splits string into three substrings. If no match is found, match substring is empty and offset past match is -1. For example, use a regular expression of + and use the following string as the input: VOLTS DC+1.22863E+1.

The *Match Pattern* function returns a before substring of VOLTS DC, a match substring of +, an after substring of 1.22863E+1, and an offset past match of 9. A regular expression requires a specific combination of characters for pattern matching.



- **Match Regular Expression**—Searches for a regular expression in the input string beginning at the offset you enter and, if it finds a match, splits the string into three substrings and any number of submatches.
- **Array To Spreadsheet String**—Converts an array of any dimension to a table in string form, containing tabs separating column elements, a platform-dependent EOL character separating rows, and, for arrays of three or more dimensions, headers separating pages.
- **Build Text**—Creates an output string from a combination of text and parameterized inputs. If the input is not a string, this Express VI converts the input into a string based on the configuration of the Express VI.
- **Carriage Return Constant**—Consists of a constant string containing the ASCII CR value.
- **Empty String Constant**—Consists of a constant string that is empty (length zero).
- **End of Line Constant**—Consists of a constant string containing the platform-dependent end-of-line value.
- **Format Date/Time String**—Displays a time stamp value or a numeric value as time in the format you specify using time format codes. Time format codes include the following: %a (abbreviated weekday name), %b (abbreviated month name), %c (locale-specific date/time), %d (day of month), %H (hour, 24-hour clock), %I (hour, 12-hour clock), %m (month number), %M (minute), %p (a.m./p.m. flag), %S (second), %x (locale-specific date), %X (locale-specific time), %y (year within century), %Y (year including century), and %<digit>u (fractional seconds with <digit> precision).
- **Format Into String**—Formats string path, enumerated type, time stamp, Boolean, or numeric data as text.
- **Line Feed Constant**—Consists of a constant string containing the ASCII LF value.
- **Replace Substring**—Inserts, deletes, or replaces a substring at the offset you specify in string.
- **Scan From String**—Scans the input string and converts the string according to format string.
- **Search and Replace String**—Replaces one or all instances of a substring with another substring.
- **Space Constant**—Use this constant to supply a one-character space string to the block diagram.
- **Spreadsheet String To Array**—Converts the spreadsheet string to an array of the dimension and representation of array type. This function works for arrays of strings and arrays of numbers.

- **String Constant**—Use this constant to supply a constant text string to the block diagram.
- **Tab Constant**—Consists of a constant string containing the ASCII HT (horizontal tab) value.
- **To Lower Case**—Converts all alphabetic characters in string to lowercase characters. Evaluates all numbers in string as ASCII codes for characters. This function does not affect non-alphabetic characters.
- **To Upper Case**—Converts all alphabetic characters in string to uppercase characters. Evaluates all numbers in string as ASCII codes for characters. This function does not affect non-alphabetic characters.
- **Trim Whitespace**—Removes all white space (spaces, tabs, carriage returns, and linefeeds) from the beginning, end, or both ends of string.

### 9.3.1 Converting Numeric Values to Strings with the Build Text Express VI

Use the Build Text Express VI to convert numeric values into strings. The Build Text Express VI, located on the *Functions»Output* palette, concatenates an input string. If the input is not a string, this Express VI converts the input into a string based on the configuration of the Express VI. When you place the Build Text Express VI on the block diagram, the *Configure Build Text* dialog box appears.

The dialog box in Figure 9.2 shows the Express VI configured to accept one input, *voltage*, and change it to a fractional number with a precision of 4. The input concatenates on the end of the string *Voltage* is. A space has been added to the end of the *Voltage* is string. This configuration produces the block diagram. A probe has been added to view the value of the output string. The Build Text Express VI concatenates the *Beginning Text* input, in this case the *voltage* value, at the end of the configured text.

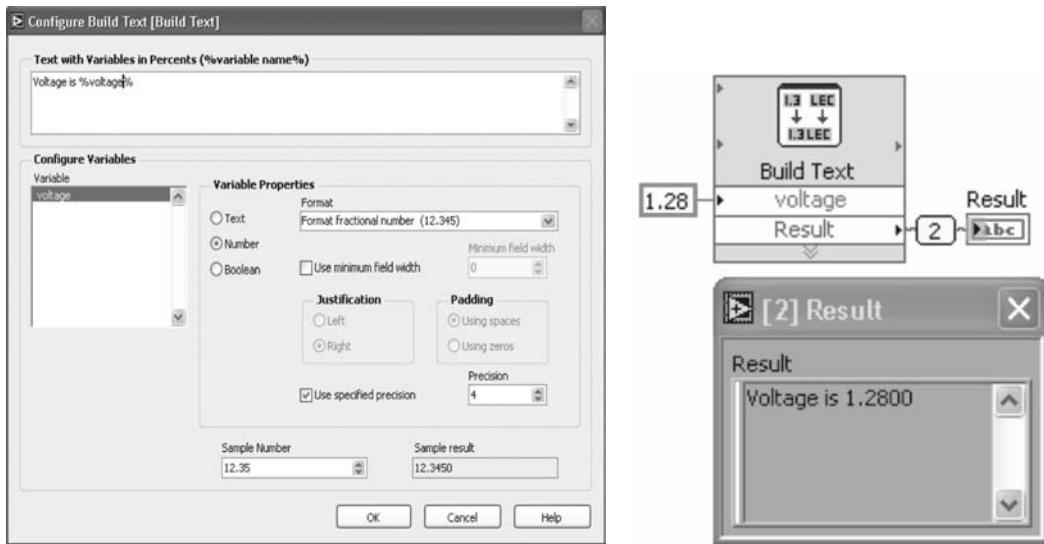
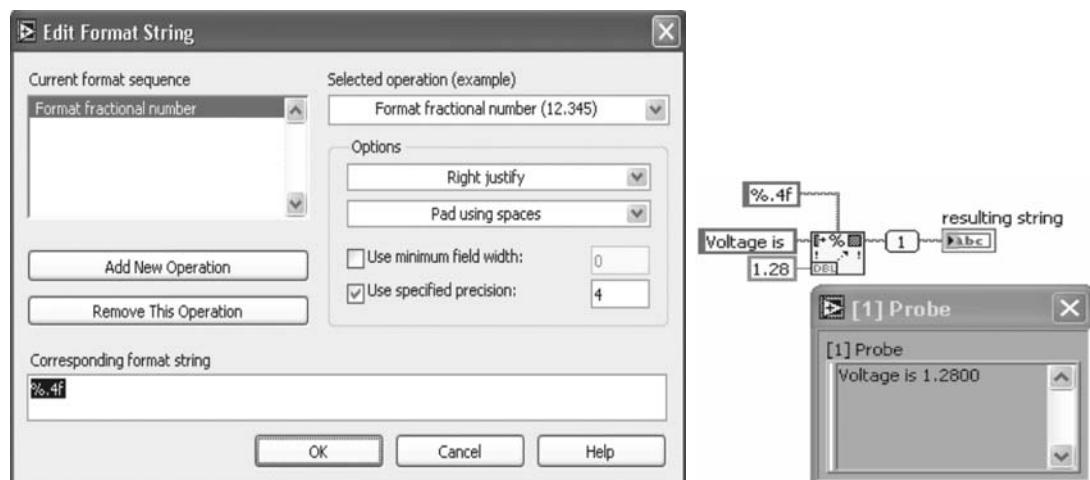


Figure 9.2 Build Text Express VI.

### 9.3.2 Converting Strings to Numeric Values with the Scan From String Function

The *Scan From String* function converts a string containing valid numeric characters, such as 0–9, +, –, e, E, and period (.), to a numeric value. This function scans the input string and converts the string according to format string. Use this function when you know the exact format of the input text. This function can scan input string into various data types, such as numeric or Boolean, based on the format string. Resize the function to increase the number of outputs.

For example as shown in Figure 9.3 use a format string of %f, an initial search location of 8, and VOLTS DC+1.28E+2 as the input string, to produce an output of 128. Change the precision of the output by changing the precision of the indicator. In format string, % begins the format specifier and f indicates a floating-point numeric with fractional format. Right-click the function and select Edit Scan String from the shortcut menu to create or edit a format string. The *Edit Scan String* dialog box shows a configuration for the format string %4f.



**Figure 9.3** Scan From String function.

## 9.4 EDITING, FORMATTING AND PARSING STRINGS

Use the *String* functions to edit strings in ways similar to the following:

- Search for, retrieve, and replace characters or substrings within a string.
- Change all text in a string to upper case or lower case.
- Find and retrieve matching patterns within a string.
- Retrieve a line from a string.
- Rotate and reverse text within a string.
- Concatenate two or more strings.
- Delete characters from a string.

To use data in another VI, function, or application, you often must convert the data to a string and then format the string in a way that the VI, function, or application can read. Microsoft Excel expects strings that include delimiters, such as tabs, commas, or blank spaces. Excel uses the

delimiter to segregate numbers or words into cells. To write a 1D array of numeric values to a spreadsheet using the *Write to Binary File* function, you must format the array into a string and separate each numeric with a delimiter, such as a tab. To write an array of numeric values to a spreadsheet using the VI, you must format the array with the *Array To Spreadsheet String* function and specify a format and a delimiter. Use the File I/O VIs and functions to save strings to text and spreadsheet files. Use the *String* functions to perform tasks similar to the following:

- Concatenate two or more strings.
- Extract a subset of strings from a string.
- Convert data into strings.
- Format a string for use in a word processing or spreadsheet application.

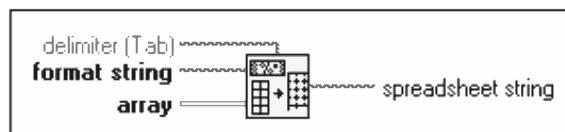
## 9.5 FORMATTING STRINGS

To use data in another VI, function, or application, you often must convert the data to a string and then format the string in a way that the VI, function, or application can read. For example, Microsoft Excel expects strings that include delimiters, such as tab, commas, or blank spaces. Excel uses the delimiter to segregate numbers or words into cells. In many cases, you must enter one or more format specifiers in the *format string* parameter of a *String* function to format a string. The format specifier is the code that indicates how to format a string. The following functions format strings:

- Array To Spreadsheet String
- Spreadsheet String To Array
- Scan From String
- Format Into String
- Format Value
- Scan Value
- Scan From File
- Format Into File

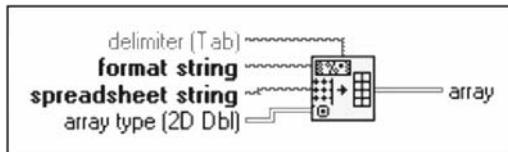
### 9.5.1 Array to Spreadsheet String

It converts an array of any dimension to a table in string form, containing tabs separating column elements, a platform-dependent EOL character separating rows, and, for arrays of three or more dimensions, headers separating pages as shown.



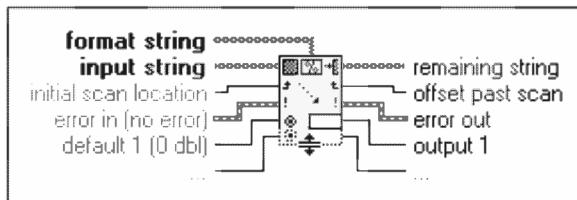
### 9.5.2 Spreadsheet String to Array

It converts the spreadsheet string to an array of the dimension and representation of array type as shown. This function works for arrays of strings and arrays of numbers. The connector pane displays the default data types for this polymorphic function.



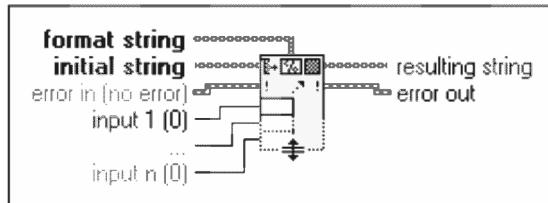
### 9.5.3 Scan From String

It scans the input string and converts the string according to format string. Use this function when you know the exact format of the input. The input can be string path, enumerated type, time stamp, or numeric data. Alternatively, you can use the *Scan From File* function to scan text from a file. The connector pane displays the default data types for this polymorphic function.



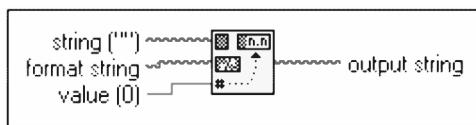
### 9.5.4 Format into String

Formats string path, enumerated type, time stamp, Boolean, or numeric data as text. Use the *Format Into File* function to format data as text and write the text to a file.



### 9.5.5 Format Value

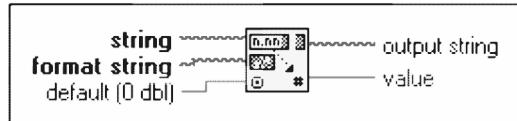
It converts a number into a regular string according to the format specified in format string and appends this to string as shown. The connector pane displays the default data types for this polymorphic function.



### 9.5.6 Scan Value

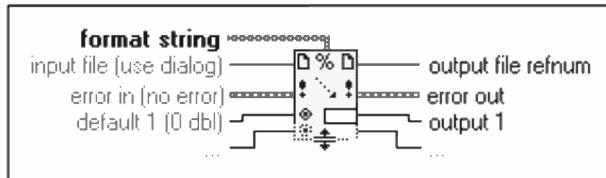
It converts characters at the beginning of string to the data type represented by default, according to the conversion codes in format string, and returns the converted number in value and the remainder

of string after the match in output string. The connector pane displays the default data types for this polymorphic function.



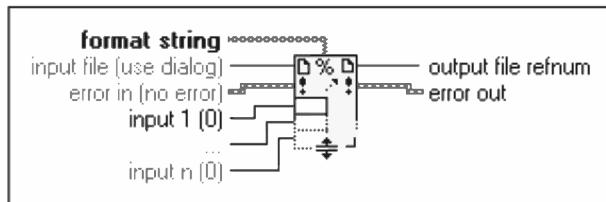
### 9.5.7 Scan from File

It scans text in a file for string, numeric, path, and Boolean data, converts the text to a data type, and returns a duplicated refnum and the converted outputs in the order scanned. You can use this function to read all the text in the file. However, you cannot use this function to determine a starting point for the scan. To do so, use the *Read From Text File* function and the *Scan From String* function.



### 9.5.8 Format into File

It formats string, numeric, path, or Boolean data as text and writes the text to a file.



## 9.6 CONFIGURING STRING CONTROLS AND INDICATORS

To configure a string control or indicator, right-click the object and select from the following shortcut menu items:

- Normal Display
- Backslash ('') Codes Display
- Password Display
- Hex Display
- Limit to Single Line
- Update Value while Typing
- Enable Wrapping

### 9.6.1 Normal Display

Right-click a string control or indicator and select *Normal Display* from the shortcut menu to display all characters as typed, with the exception of non-displayable characters. Non-displayable characters generally appear as boxes.

### 9.6.2 Backslash ('\\') Codes Display

Right-click a string control or indicator and select '*\*' *Codes Display* from the shortcut menu to instruct LabVIEW to interpret characters that immediately follow a backslash (*\*) as a code for non-displayable characters. The backslash mode is useful for debugging VIs and for sending non-displayable characters to instruments, serial ports and other devices.

### 9.6.3 Password Display

Right-click a string control or indicator and select *Password Display* from the shortcut menu to display an asterisk (\*) for each character you enter into a string control, including spaces. When you read the string data from the block diagram, you read the actual data the user entered. If you try to copy data from the control, LabVIEW copies only the \* characters.

### 9.6.4 Hex Display

Right-click a string control or indicator and select *Hex Display* from the shortcut menu to display the ASCII value of each character in hex instead of the character itself. The *Hex Display* item is useful for debugging and communicating with instruments.

### 9.6.5 Limit to Single Line

Right-click a string control or indicator and select *Limit to Single Line* from the shortcut menu to prevent the user from entering a carriage return in a string control or indicator. However, if you select this option, the user can copy multiline strings and paste them in the string control or indicator. You also can use the *Limit To Single Line?* property to programmatically prevent a user from entering a carriage return.

### 9.6.6 Update Value while Typing

Right-click a string control or indicator and select *Update Value while Typing* from the shortcut menu to update the value of a control as the user enters characters instead of waiting until the user presses the <Enter> key or otherwise ends text entry. Use this option to check the correctness of the input or to give user feedback. You also can use the *Update while Typing?* property to update the value programmatically.

### 9.6.7 Enable Wrapping

To disable word wrapping in a string control or indicator, right-click the string and remove the checkmark next to the *Enable Wrapping* shortcut menu item. Disabling word wrapping causes the string to wrap at line breaks instead. If you want to disable word wrapping, the string must be in

normal display mode. After you disable word wrapping, you can display a horizontal scroll bar by right-clicking the string control or indicator and selecting *Visible Items»Horizontal Scrollbar* from the shortcut menu. You also can use the *Enable Wrapping property* to disable word wrapping programmatically.

## 9.7 BASICS OF FILE INPUT/OUTPUT

File I/O records or reads data in a file. File I/O operations pass data to and from files. Use the file I/O VIs and functions located on the *Functions»All Functions»File I/O* palette to handle all aspects of file I/O, including the following:

- Opening and closing data files
- Reading data from and writing data to files
- Reading from and writing to spreadsheet-formatted files
- Moving and renaming files and directories
- Changing file characteristics
- Creating, modifying and reading configuration files

## 9.8 CHOOSING A FILE I/O FORMAT

LabVIEW can use or create the following file formats: Binary, ASCII, LVM, and TDM.

- **Binary**—Binary files are the underlying file format of all other file formats.
- **ASCII**—An ASCII file is a specific type of binary file that is a standard used by most programs. It consists of a series of ASCII codes. ASCII files are also called text files.
- **LVM**—The LabVIEW measurement data file (.lvm) is a tab-delimited text file you can open with a spreadsheet application or a text-editing application. The .lvm file includes information about the data, such as the date and time the data was generated. This file format is a specific type of ASCII file created for LabVIEW.
- **TDM**—This file format is a specific type of binary file created for National Instruments products. It actually consists of two separate files: an XML section contains the data attributes and a binary file for the waveform.

The VIs on the File I/O palette you use depend on the format of the files. You can read data from or write data to files in three formats—text, binary and datalog. The format you use depends on the data you acquire or create and the applications that will access that data. Use the following basic guidelines to determine which format to use:

- If you want to make your data available to other applications, such as Microsoft Excel, use text files because they are the most common and the most portable.
- If you need to perform random access file reads or writes or if speed and compact disk space are crucial, use binary files because they are more efficient than text files in disk space and in speed.
- If you want to manipulate complex records of data or different data types in LabVIEW, use datalog files because they are the best way to store data if you intend to access the data only from LabVIEW and you need to store complex data structures.

### 9.8.1 Use of Text Files

Use text format files for your data to make it available to other users or applications, if disk space and file I/O speed are not crucial, if you do not need to perform random access reads or writes, and if numeric precision is not important. Text files are the easiest format to use and to share. Almost any computer can read from or write to a text file. A variety of text-based programs can read text-based files. Most instrument control applications use text strings. Store data in text files when you want to access it from another application, such as a word processing or spreadsheet application. To store data in text format, use the *String* functions to convert all data to text strings. Text files can contain information of different data types. Text files typically take up more memory than binary and datalog files if the data is not originally in text form, such as graph or chart data, because the ASCII representation of data usually is larger than the data itself. For example, you can store the number  $-123.4567$  in 4 bytes as a single-precision floating-point number. However, its ASCII representation takes 9 bytes, one for each character.

In addition, it is difficult to randomly access numeric data in text files. Although each character in a string takes up exactly 1 byte of space, the space required to express a number as text typically is not fixed. To find the ninth number in a text file, LabVIEW must first read and convert the preceding eight numbers. You might lose precision if you store numeric data in text files. Computers store numeric data as binary data, and typically you write numeric data to a text file in decimal notation. A loss of precision might occur when you write the data to the text file. Loss of precision is not an issue with binary files. Use the File I/O VIs and functions to read from or write to text files and to read from or write to spreadsheet files.

### 9.8.2 Use of Binary Files

Storing binary data, such as an integer, uses a fixed number of bytes on disk. For example, storing any number from 0 to 4 billion in binary format, such as 1, 1,000, or 1,000,000, takes up 4 bytes for each number. Use binary files to save numeric data and to access specific numbers from a file or randomly access numbers from a file. Binary files are machine readable only, unlike text files which are human readable. Binary files are the most compact and fastest format for storing data. You can use multiple data types in binary files, but it is uncommon.

Binary files are more efficient because they use less disk space and because you do not need to convert data to and from a text representation when you store and retrieve data. A binary file can represent 256 values in 1 byte of disk space. Often, binary files contain a byte-for-byte image of the data as it was stored in memory, except for cases like extended and complex numeric values. When the file contains a byte-for-byte image of the data as it was stored in memory, reading the file is faster because conversion is not necessary. Text and binary files are both known as byte stream files, which means they store data as a sequence of characters or bytes. Use the File I/O VIs and functions to read from and write to binary files. Consider using the binary file functions if you want to read numeric data from or write numeric data to a file or if you want to create text files for use on multiple operating systems.

### 9.8.3 Use of Datalog Files

Use datalog files to access and manipulate data only in LabVIEW and to store complex data structures quickly and easily. A datalog file stores data as a sequence of identically structured

records, similar to a spreadsheet, where each row represents a record. Each record in a datalog file must have the same data types associated with it. LabVIEW writes each record to the file as a cluster containing the data to store. However, the components of a datalog record can be any data type which you determine when you create the file. For example, you can create a datalog whose record data type is a cluster of a string and a number. Then, each record of the datalog is a cluster of a string and a number. However, the first record could be (“abc”,1), while the second record could be (“xyz”,7).

Using datalog files requires little manipulation which makes writing and reading much faster. It also simplifies data retrieval because you can read the original blocks of data back as a record without having to read all records that precede it in the file. Random access is fast and easy with datalog files because all you need to access the record is the record number. LabVIEW sequentially assigns the record number to each record when it creates the datalog file. You can access datalog files from the front panel and from the block diagram. LabVIEW writes a record to a datalog file each time the associated VI runs. You cannot overwrite a record after LabVIEW writes it to a datalog file. When you read a datalog file, you can read one or more records at a time.

## 9.9 LabVIEW DATA DIRECTORY

You can use the default LabVIEW Data directory to store the data files LabVIEW generates, such as .lvm or .txt files. LabVIEW installs the LabVIEW Data directory in the default file directory for your operating system to help you organize and locate the data files LabVIEW generates. By default, the Write LabVIEW Measurement File Express VI stores the .lvm files it generates in this directory, and the Read LabVIEW Measurement File Express VI reads from this directory. The Default Data Directory constant, shown at left, and the Default Data Directory property also return the LabVIEW Data directory by default. Select *Tools»Options* and select *Paths* from the top pull-down menu to specify a different default data directory. The default data directory differs from the default directory which is the directory you specify for new VIs, custom controls, VI templates, or other LabVIEW documents you create.

## 9.10 FILE I/O VIs

A typical file I/O operation involves the following process,

1. Create or open a file. Indicate where an existing file resides or where you want to create a new file by specifying a path or responding to a dialog box to direct LabVIEW to the file location. After the file opens, a refnum represents the file.
2. Read from or write to the file.
3. Close the file.

File I/O VIs and some File I/O functions, such as the Read from Text File and Write to Text File functions, can perform all three steps for common file I/O operations. The VIs and functions designed for multiple operations might not be as efficient as the functions configured or designed for individual operations. Many File I/O VIs and functions contain flow-through parameters, typically a refnum or path, which return the same value as the corresponding input parameter. If

you are writing to a file in a loop, use low level file I/O VIs. If you are writing to a file in a single operation, use the high-level file I/O VIs if you prefer.

### 9.10.1 Disk Streaming with Low-Level Functions

You also can use File I/O functions for disk streaming operations which save memory resources by reducing the number of times the function interacts with the operating system to open and close the file. Disk streaming is a technique for keeping files open while you perform multiple write operations, for example, within a loop. Wiring a path control or a constant to the *Write to Text File* function, the *Write to Binary File* function, or the *Write to Spreadsheet File*, VI adds the overhead of opening and closing the file each time the function or VI executes. VIs can be more efficient if you avoid opening and closing the same files frequently. To avoid opening and closing the same file, you need to pass a refnum to the file into the loop. When you open a file, device or network connection, LabVIEW creates a refnum associated with that file, device or network connection. All operations you perform on open files, devices or network connections use the refnums to identify each object.

### 9.10.2 High Level File I/O

High-level File I/O VIs include the following:

**Write to Spreadsheet File**—Converts a 2D or 1D array of single-precision numbers to a text string and writes the string to a new ASCII file or appends the string to an existing file. You also can transpose the data. The VI opens or creates the file before writing to it and closes it afterwards. You can use this VI to create a text file readable by most spreadsheet applications as illustrated in Figure 9.4.

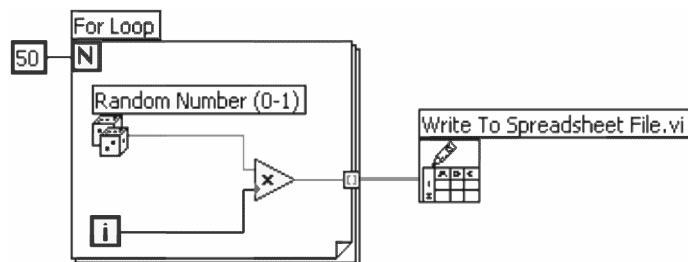


Figure 9.4 Write to Spreadsheet File.

**Read From Spreadsheet File**—Reads a specified number of lines or rows from a numeric text file beginning at a specified character offset and converts the data to a 2D single-precision array of numbers. The VI opens the file before reading from it and closes it afterwards. You can use this VI to read a spreadsheet file saved in text format.

**Write to/Read from Measurement File**—Express VIs that write or read data to or from a text-based measurement file (.lvm) or a binary measurement file (.tdm) format. It includes the open, write, close and error handling functions. This file handles formatting the string with either a tab or comma delimiter. The *Merge Signals* function is used to combine data into the dynamic data type as illustrated in Figure 9.5.

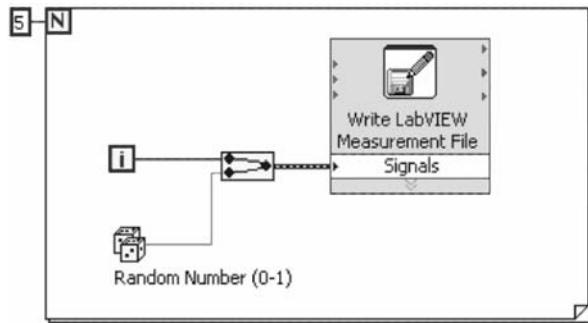


Figure 9.5 Write LabVIEW Measurement File.

## 9.11 CREATING A RELATIVE PATH

A relative path describes the location of a file or directory relative to an arbitrary location in the file system. An absolute path describes the location of a file or directory starting from the top level of the file system. Relative paths are also referred to as symbolic paths. Use relative paths in VIs to avoid having to rework the paths when you build an application or run the VI on a different computer.

Complete the following steps to create a relative path.

1. Place the *Default Directory* constant on the block diagram.
2. Place the *Build Path* function on the block diagram.
3. Wire the *Default Directory* constant to the base path input of the *Build Path* function.
4. Right-click the name or relative path input of the *Build Path* function and select *Create»Constant*.
5. Use the labeling tool to double-click the name or relative path constant and enter the filename or relative path to the file. Use the syntax appropriate for your operating system.
  - Windows A path consists of the drive name, followed by a colon, followed by backslash-separated directory names, followed by the filename. An example is C:\DATADIR\TEST1 for a file named TEST1 in the directory DATADIR on the C drive. You also can drag a path from Windows Explorer and place it in the path constant.
  - Mac OS A path consists of the drive name, followed by colon-separated folder names, followed by the filename. An example is HardDrive:DataFolder:Test1 for a file named Test1 in the folder DataFolder on the volume named HardDrive.
  - Linux A path consists of slash-separated directory names followed by the filename. An example is /usr/datadirectory/test1 for a file named test1 in the directory /usr/datadirectory.

## SUMMARY

- Strings group sequences of ASCII characters. Use the string control and indicator to simulate text entry boxes and labels.

- To minimize the space that strings object occupies right-click the object and select *Show Scrollbar* from the shortcut menu.
- Use the *String* functions located on the *Functions»All Functions»String* palette to edit and manipulate strings on the block diagram.
- Use the *Build Text Express VI* to convert a numeric value to a string.
- Use the *Scan From String* function to convert a string to a numeric value.
- Right-click the *Scan From String* function and select *Edit Scan String* from the shortcut menu to create or edit a *format string*.
- Use the *File I/O* VIs and functions located on the *Functions»File I/O* palette to handle all aspects of file I/O.
- When writing to a file, you open, create, or replace a file, write the data and close the file. Similarly, when you read from a file, you open an existing file, read the data and close the file.
- To access a file through a dialog box, do not wire file path in the *Open/Create/Replace File VI*.
- To write data to a spreadsheet file, you must format the string as a spreadsheet string, which is a string that includes delimiters, such as tabs. Use the *Format Into File* function to format string, numeric, path and Boolean data as text and write the text to a file.

## MISCELLANEOUS SOLVED PROBLEMS

**Problem 9.1** Create a VI which consists of two string inputs. Find the length of each string input. Join the strings using *Concatenate String Function*. Find the length of the concatenated string.

**Solution** The front panel and the block diagram to solve the string problem are shown in Figures P9.1(a) and P9.1(b).

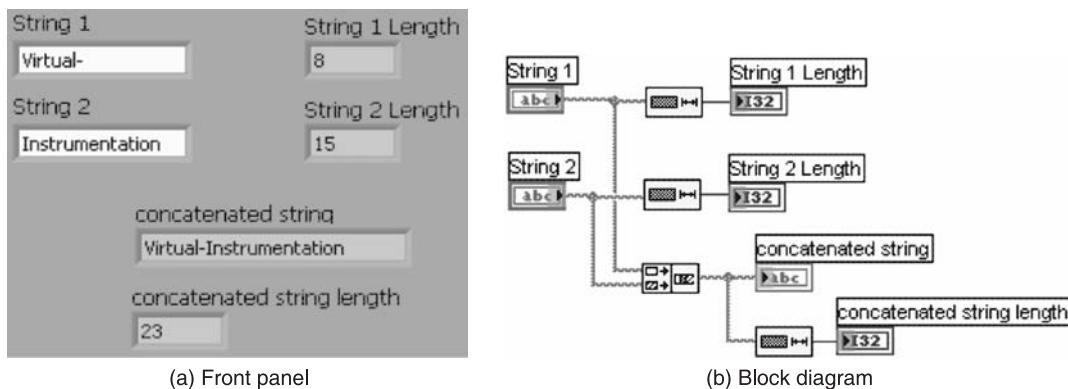


Figure P9.1

**Problem 9.2** Build a VI which gets a string input. Replace a particular word in the input string by a new word. Use the *Replace Substring* function for this.

**Solution** The front panel and the block diagram to use the string function and solve the problem are shown in Figures P9.2(a) and P9.2(b).

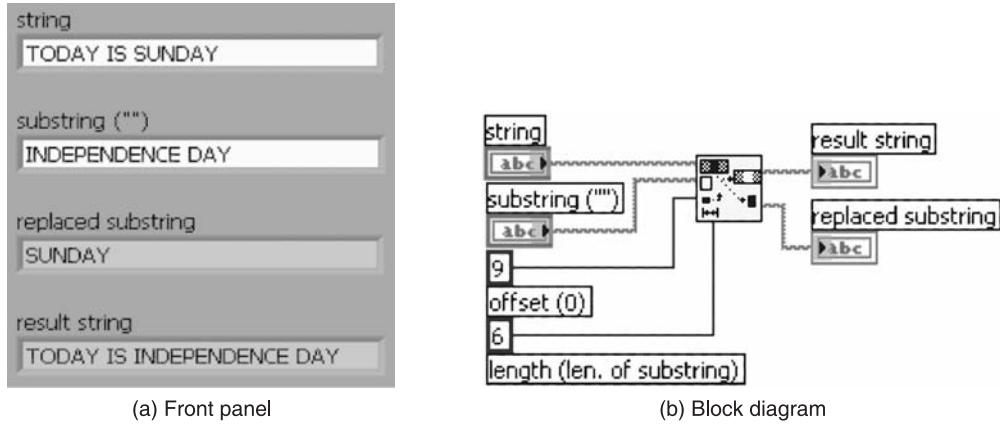


Figure P9.2

**Problem 9.3** Create a VI to format the date and time in the required format using *Format Date/Time String Function*. Get the date and time input from *Time Stamp Control*.

**Solution** Build the front panel and the block diagram to solve the problem as shown in Figures P9.3(a) and P9.3(b).

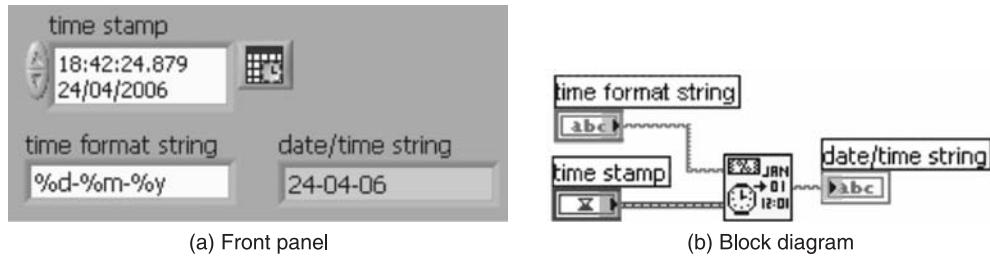


Figure P9.3

**Problem 9.4** Use *Format into String Function* to get a data in the prescribed format and add with the already available string.

**Solution** The front panel and the block diagram to solve the problem using the string function are shown in Figures P9.4(a) and P9.4(b).

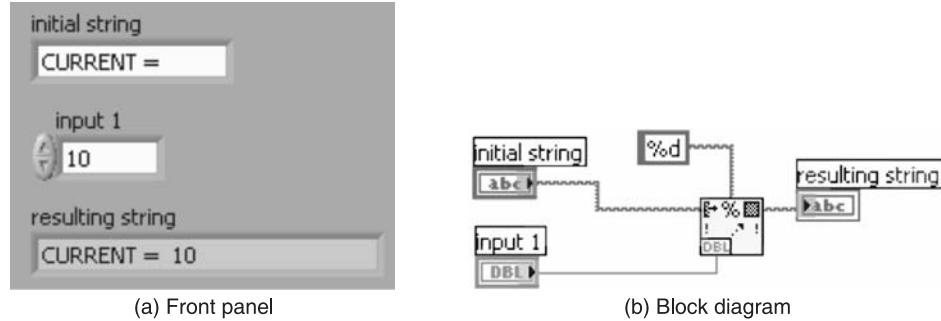


Figure P9.4

**Problem 9.5** Use the *Array to Spreadsheet String Function* to format an array of data in the spreadsheet format. Use the *Spreadsheet String to Array Function* to convert a spreadsheet string to an array format.

**Solution** The front panel and the block diagram to use the *Array to Spreadsheet String Function* are shown in Figures P9.5(a) and P9.5(b).

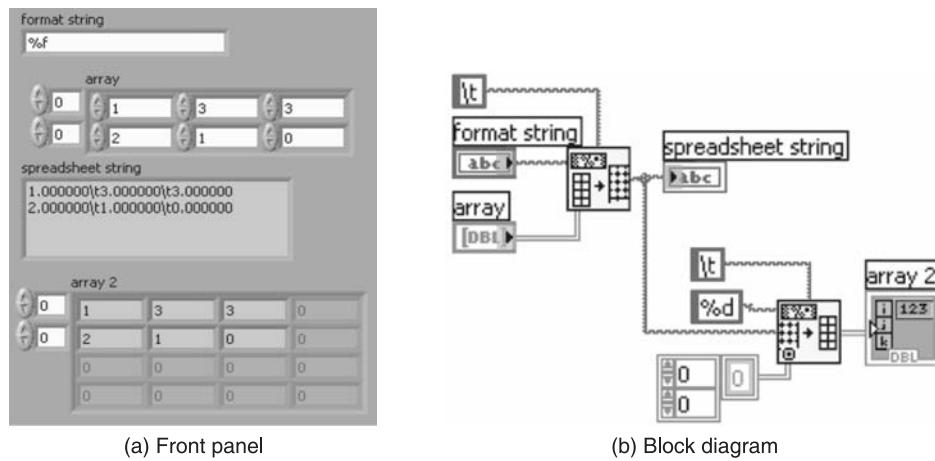
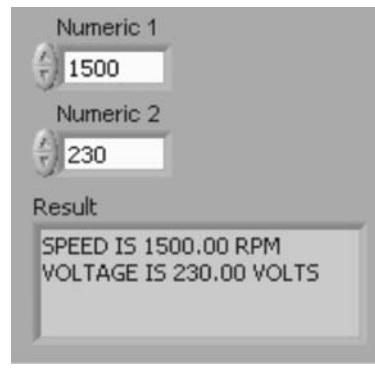


Figure P9.5

**Problem 9.6** Use Build Text Express VI to build a text by getting inputs from numeric controls. The Configure Build Text Window is shown below. Also use Prompt User for Input Express VI to get inputs to build a text.

**Solution** The front panel and the block diagram using the Build Text Express VI are shown in Figures P9.6(a) and P9.6(b). The front panel and the block diagram using the Prompt User for Input Express VI to get inputs to build a text are shown in Figures P9.6(c) and P9.6(d).



(a) Front panel

Figure P9.6 (Contd.)

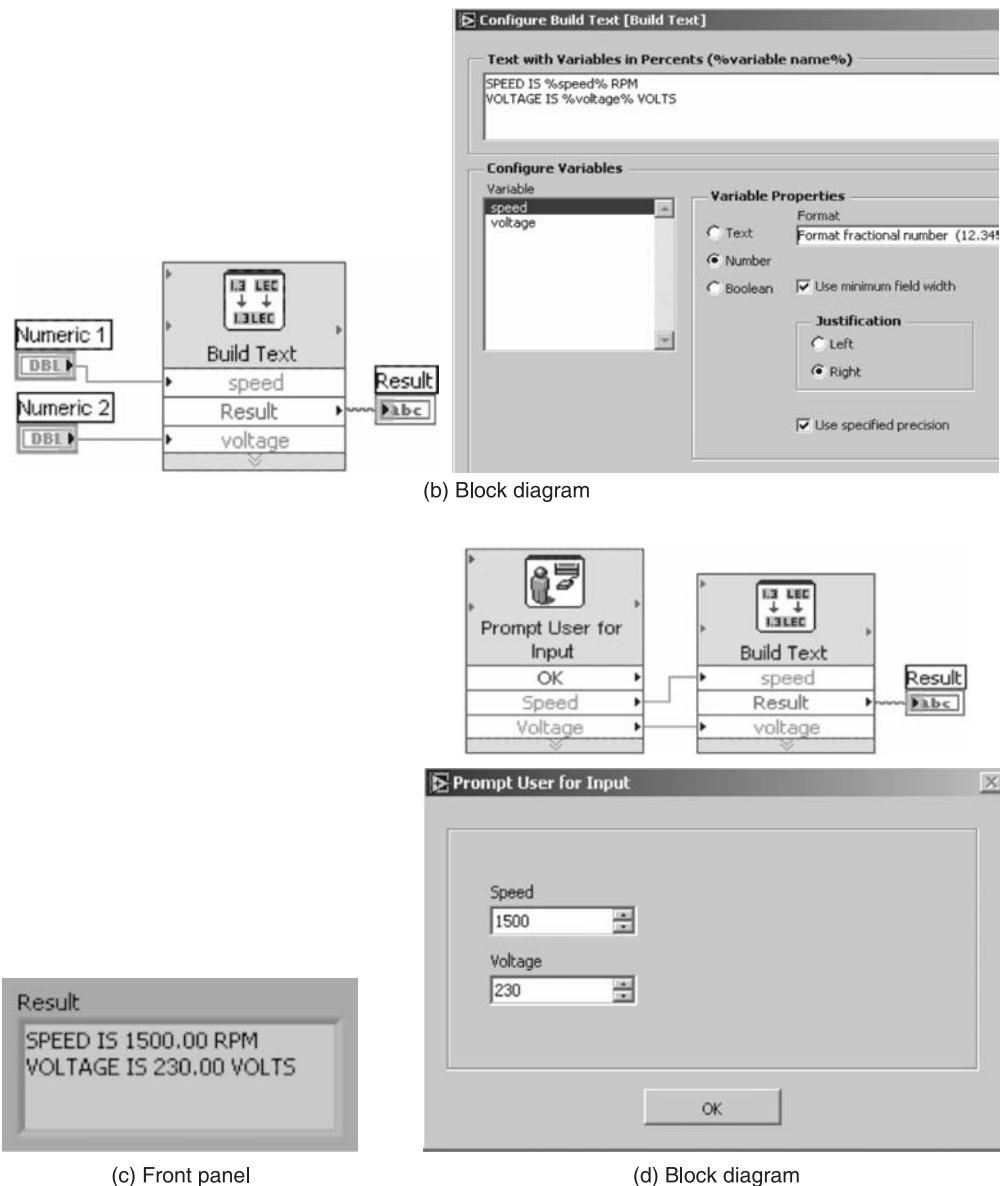
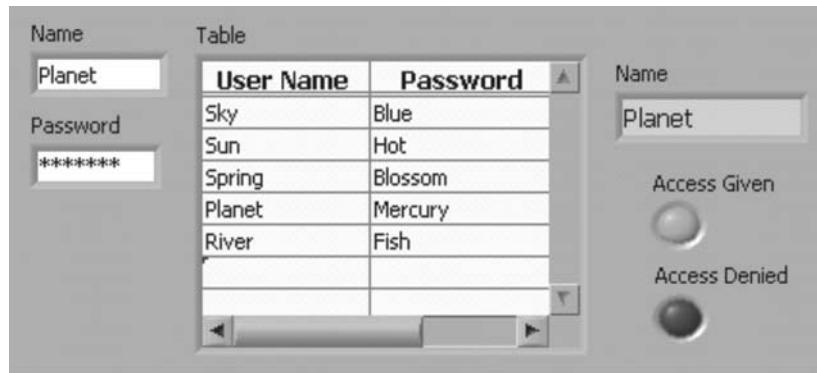


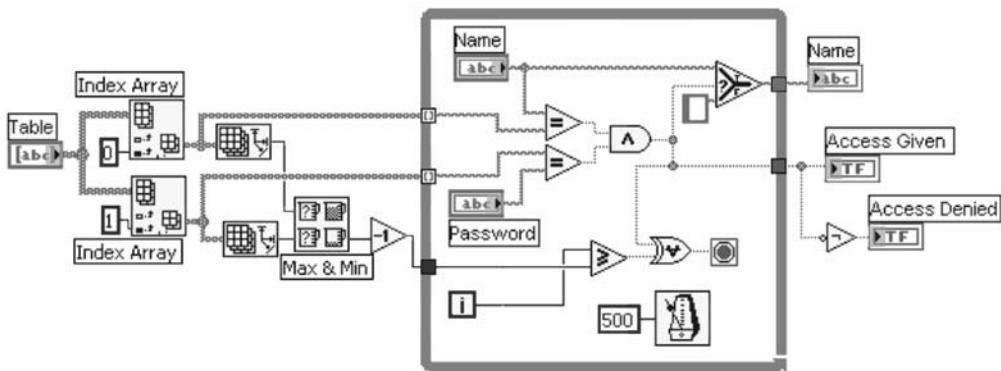
Figure P9.6

**Problem 9.7** Create a table which consists of user names and passwords. Input a user name and a password. Check whether both the user name and password match the contents of the table. If they are matched glow ‘Access Given’ LED, otherwise glow ‘Access Denied’ LED. Also display the user name.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P9.7(a) and P9.7(b).



(a) Front panel

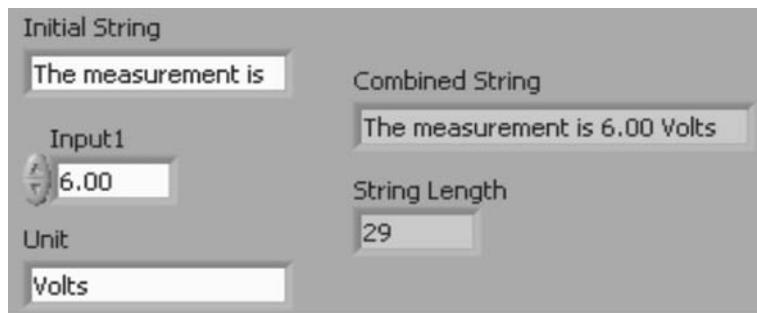


(b) Block diagram

Figure P9.7

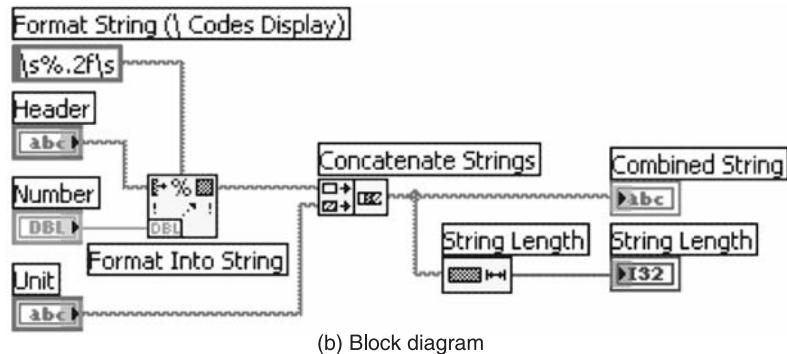
**Problem 9.8** Use the *Format Into String Function* to combine a text with a number.

**Solution** The front panel and the block diagram using the *Format Into String Function* are shown in Figures P9.8(a) and P9.8(b).



(a) Front panel

Figure P9.8 (Contd.)

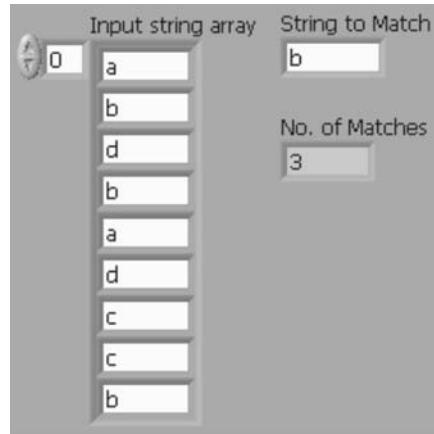


(b) Block diagram

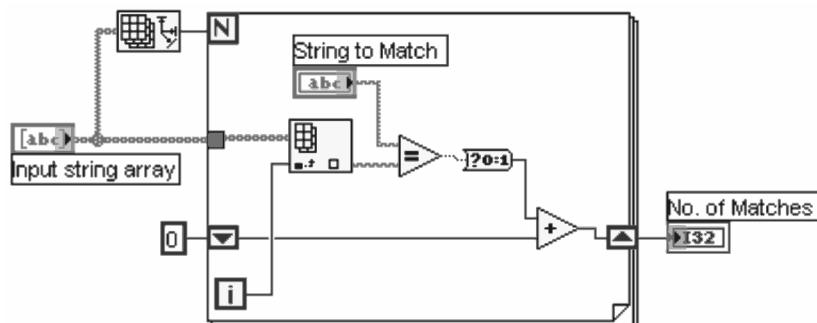
**Figure P9.8**

**Problem 9.9** Build a VI which finds the number of occurrences of a particular string in an array of strings.

**Solution** The front panel and the block diagram to solve the problem are shown in Figures P9.9(a) and P9.9(b).



(a) Front panel



(b) Block diagram

**Figure P9.9**

**Problem 9.10** Build a VI to split numbers and words available in a string. Display the splitted numbers and words in separate arrays.

**Solution** Create the front panel and the block diagram to solve the problem as shown in Figures P9.10(a) and P9.10(b).

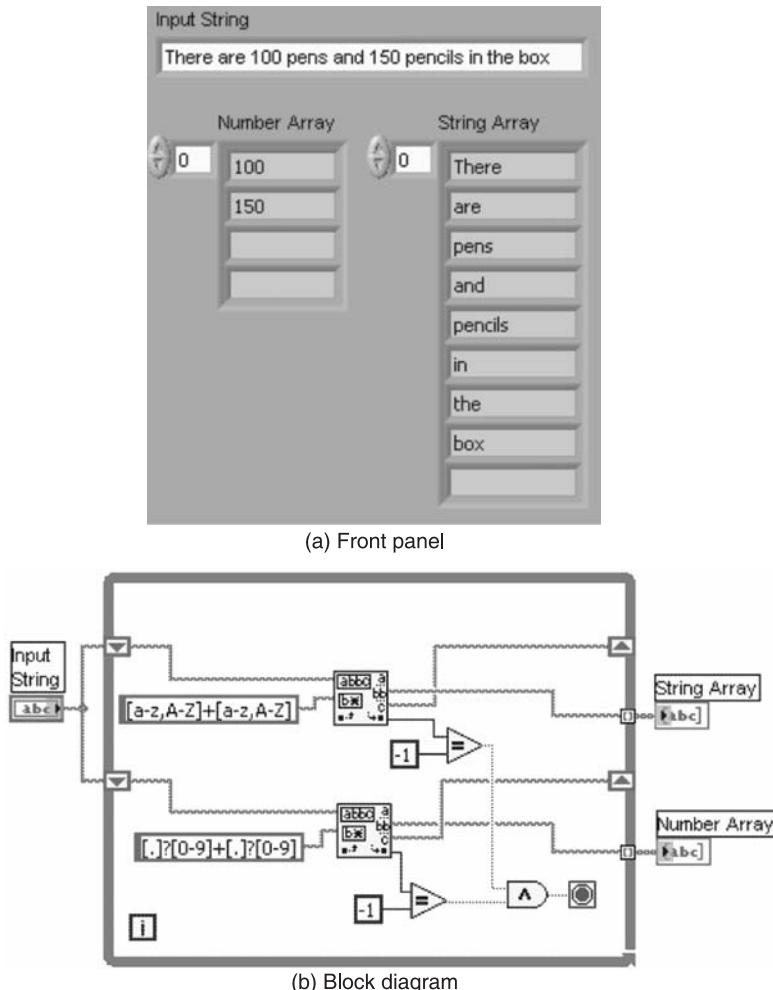
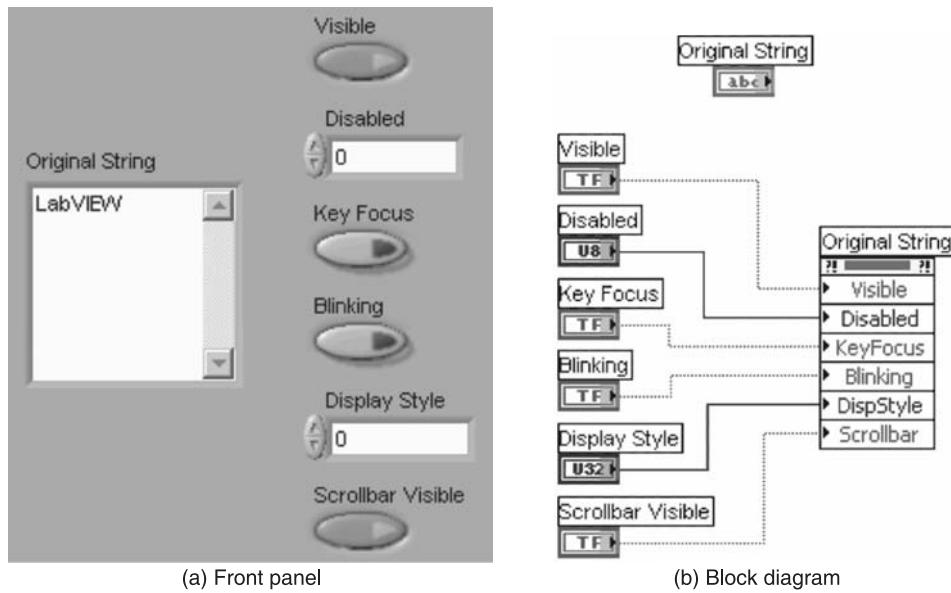


Figure P9.10

**Problem 9.11** For a string input, change the properties like visible, displayed, etc. using property nodes.

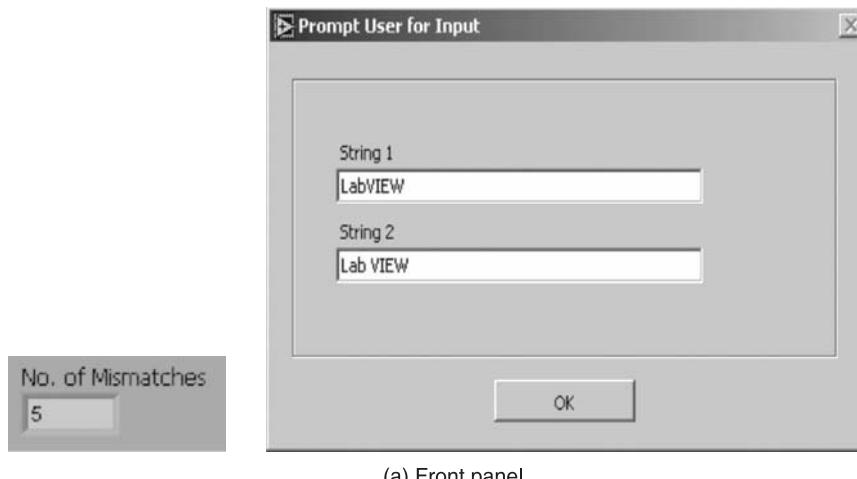
**Solution** The front panel and the block diagram to solve the problem are shown in Figures P9.11(a) and P9.11(b).



**Figure P9.11**

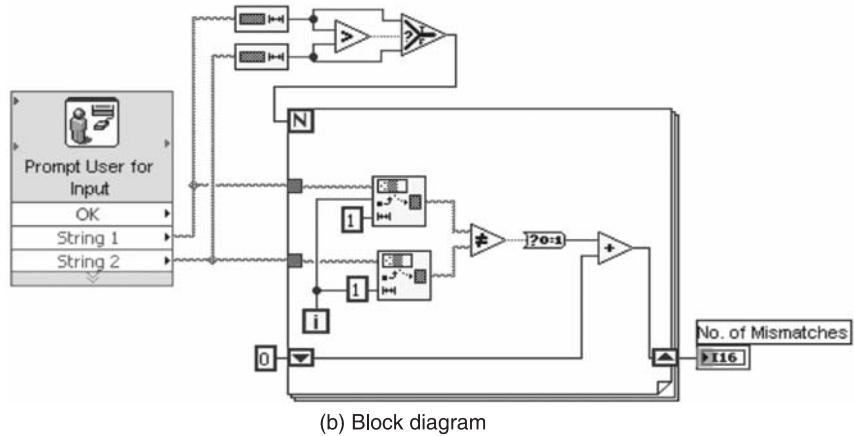
**Problem 9.12** Create a VI which prompts for user inputs to get two input strings. Compare each character of string 1 with each character of string 2. Display the number of mismatches.

**Solution** Build the front panel and the block diagram to solve the problem as shown in Figures P9.12(a) and P9.12(b).



(a) Front panel

**Figure P9.12 (Contd.)**

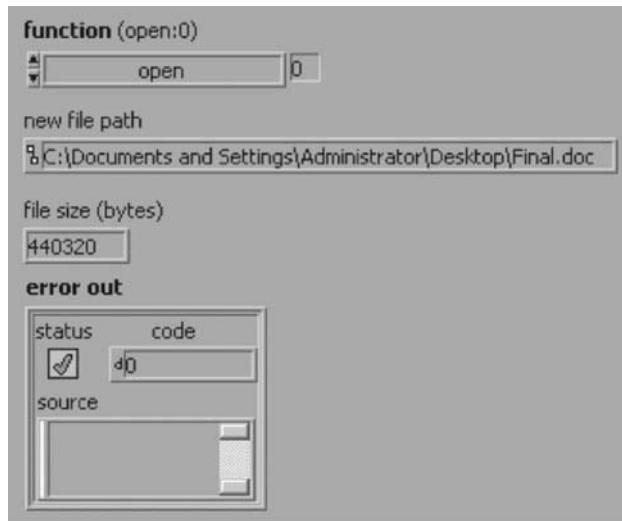


(b) Block diagram

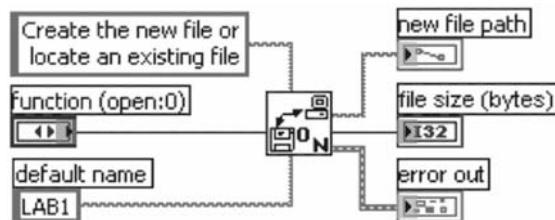
**Figure P9.12**

**Problem 9.13** Create a VI to open a file and display the size of the file (in bytes).

**Solution** The front panel and the block diagram to open and display the size of a file are shown in Figures P9.13(a) and P9.13(b).



(a) Front panel

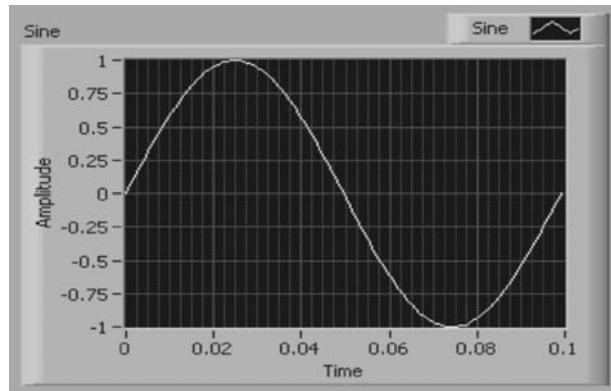


(b) Block diagram

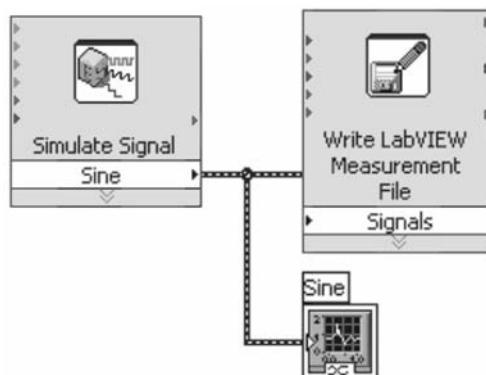
**Figure P9.13**

**Problem 9.14** Simulate a sine wave using Simulate Signal Express VI. Store the values of all the points of the sine wave in a file using Write LabVIEW Measurement File Express VI.

**Solution** The front panel and the block diagram to use Simulate Signal Express VI are shown in Figures P9.14(a) and P9.14(b).



(a) Front panel



(b) Block diagram

**Figure P9.14**

**Problem 9.15** Create a VI to open and read a particular file. Display the file path and contents of the file. Display the numeric data and strings in separate files.

**Solution** The front panel and the block diagram to create a VI to open and read a particular file are shown in Figures P9.15(a) and P9.15(b).

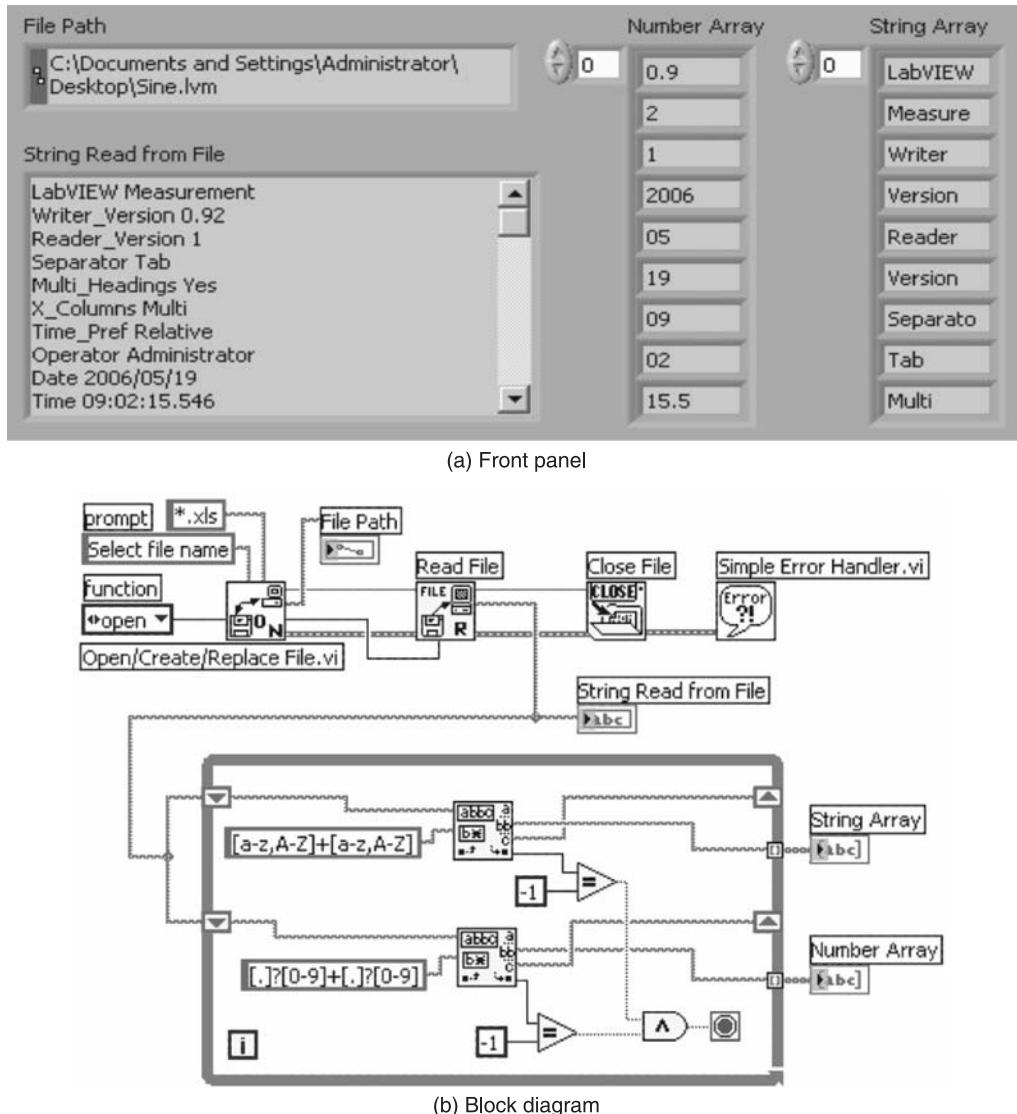
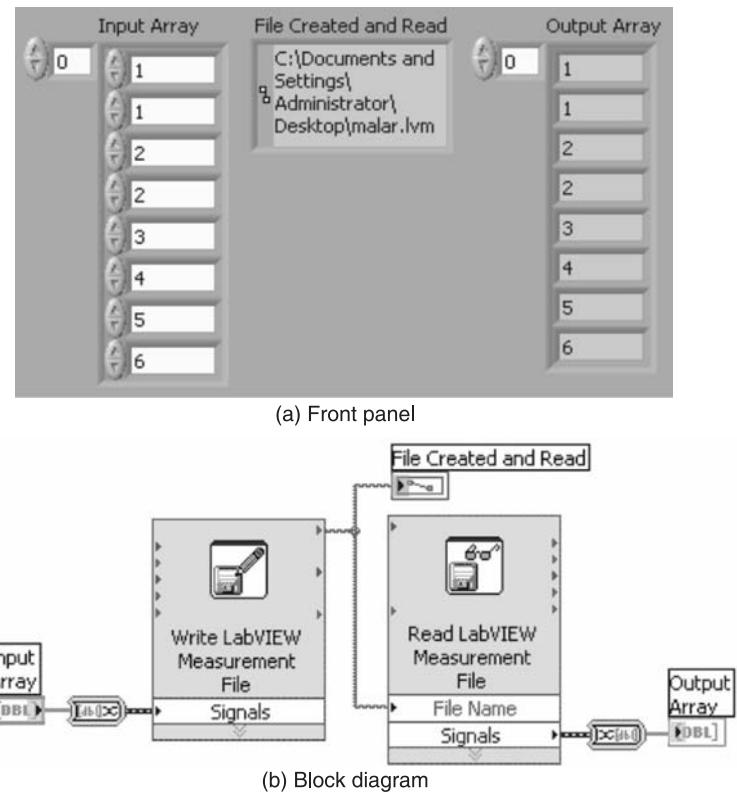


Figure P9.15

**Problem 9.16** Create an array of numeric values and write in a file using Write LabVIEW Measurement File. Read the same file using Read LabVIEW Measurement File. Display the read contents in an array indicator. Also display the file created and read.

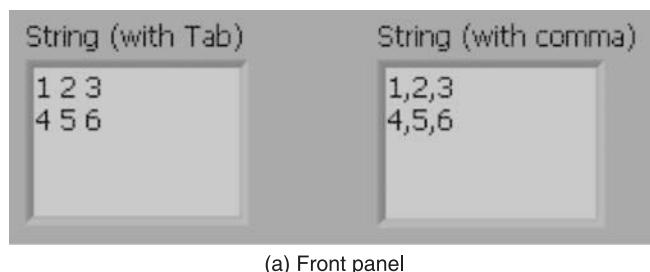
**Solution** The front panel and the block diagram to solve the problem are shown in Figures P9.16(a) and P9.16(b).



**Figure P9.16**

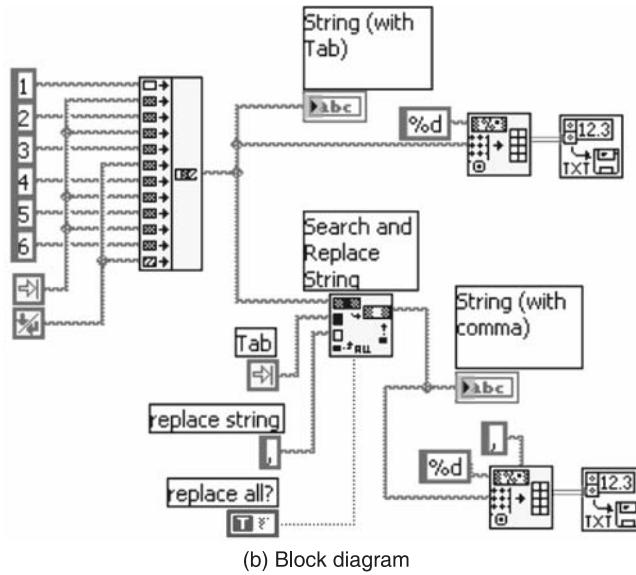
**Problem 9.17** Create numbers from 1 to 6 with tab as a delimiter. Write this data in a spreadsheet file. Replace the tab delimiter by comma ( , ) and write in another spreadsheet file.

**Solution** Build the front panel and the block diagram as shown in Figures P9.17(a) and P9.17(b) to create numbers from 1 to 6 with tab as a delimiter.



(a) Front panel

**Figure P9.17 (Contd.)**



(b) Block diagram

**Figure P9.17**

## REVIEW QUESTIONS

1. What are the common applications of strings?
2. What are string controls and indicators?
3. Explain the various display types with example messages.
4. Draw and explain how a table can be used to display a 2D array of strings.
5. How can numeric values be converted to strings with the Build Text Express VI?
6. Draw and explain some of the common string functions in LabVIEW.
7. List the functions of format strings and explain the use.
8. What are the commonly used file formats?
9. Mention the options available to write the front panel data to the file.
10. What is the difference between High Level and Low Level file functions?
11. What are the functions available to write data into a spreadsheet file?
12. List the advantages of Write LabVIEW Measurement File and Read LabVIEW Measurement File.

## EXERCISES

1. Create a VI to convert the input string to uppercase and lowercase.
2. Build a VI to replace a particular word in a string with a new word using the *Search and Replace String* function.

3. Using the *Match Pattern* function check whether a given expression is available in the input string. Also display the portion of the string available before the match and after the match, and the offset of the string after the match.
4. Build a VI to create a substring from an input string by providing the offset of the substring and length of the substring. Use the *String Subset* function for this. Also find the reversed and rotated (first character last) version of the substring.
5. Use the *Scan From String* function to find a part of the string in the prescribed format. Also display the remaining string and offset after the scan.
6. Use the *Pick Line* function to separate a particular line from a multi-line text.
7. Build a VI to find the number of matches of a particular character in a string.
8. Build a VI which gets two arrays of strings as inputs. Compare each element of array1 with all the elements of array2. In a table display 1s for matches and 0s for non-matches.
9. In the previous example display array elements along with 0s and 1s in the table.
10. In the previous example use string controls instead of string arrays.
11. Create a VI to find whether the input string is a palindrome or not.
12. Create a VI to write the contents of a string control in a file whose path is specified.
13. Generate a 2D numeric array using random numbers. Display the generated values in a waveform graph. Write the same values in a spreadsheet file. While writing in a spreadsheet file add column headings. Let the column headings be Waveform1, Waveform 2 and Waveform 3.
14. Create a VI to read a file and display its contents in a string indicator.
15. Read the same file which was read in the previous example using low level file functions.
16. Read a spreadsheet file which consists of multiple columns. Display the contents in a two-dimensional array.



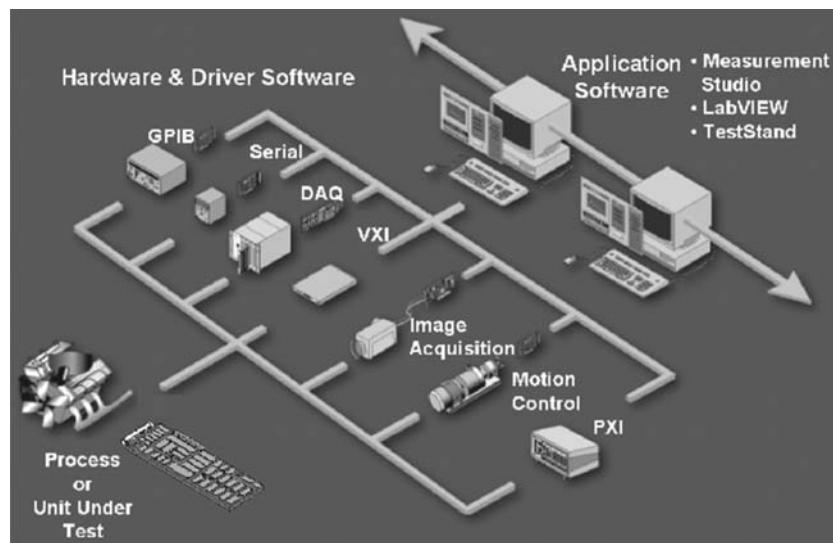
# INSTRUMENT CONTROL

## 10.1 INTRODUCTION

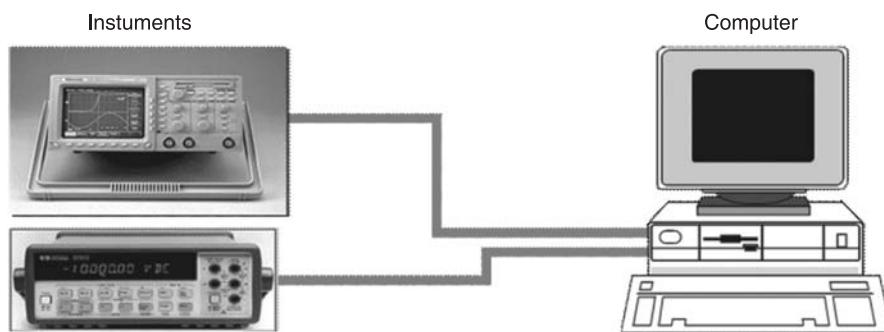
Computer-based measurement and automation is popular because our PC provides the platform we need to make our measurement and automation systems dependable and efficient. Its extensive processing capabilities empower us to create flexible solutions based on industry standards. With this flexibility, we can adjust our application specifications more easily than with traditional tools.

National Instruments software, including LabVIEW and Measurement Studio, delivers PC-based data analysis, connectivity, and presentation power to new levels in measurement and automation applications. National Instruments hardware and software connect the computer to your application. By providing an extensive hardware selection, including data acquisition and signal conditioning devices, instrument control interfaces (such as GPIB, Serial, VXI and PXI), image acquisition, motion control and industrial communications interfaces, National Instruments offers the widest range of solutions for practically any measurement as shown in Figure 10.1.

This chapter describes instrument control of stand-alone instruments using a GPIB or serial interface. Use LabVIEW to control and acquire data from instruments with the Instrument I/O Assistant, the VISA API, and instrument drivers. If you choose industry-standard control technologies, you are not limited to the type of instrument you can control. You can mix and match instruments from various categories as shown in Figure 10.2. The most common categories of instrument interfaces are GPIB, serial, modular instruments and PXI modular instruments. Additional types of instruments include image acquisition, motion control, USB, Ethernet, parallel port, NI-CAN and other devices.



**Figure 10.1** Computer-based measurement and automation.



**Figure 10.2** PC control of instruments.

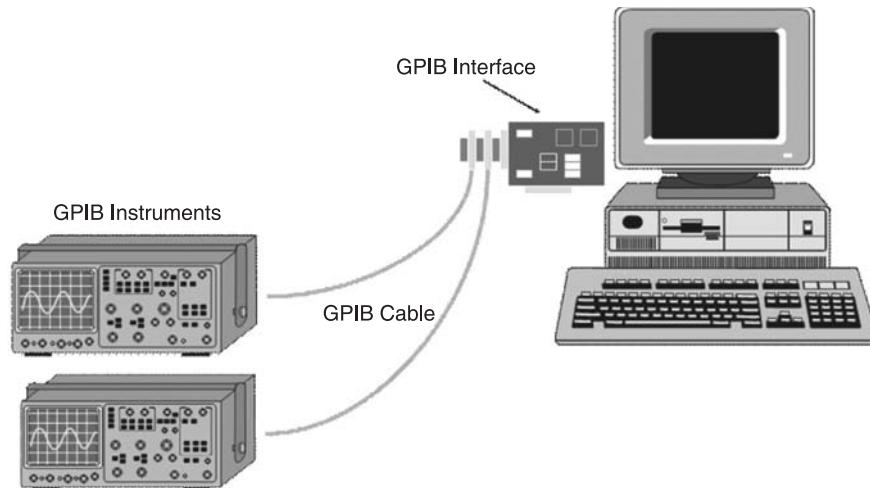
When you use PCs to control instruments, you need to understand properties of the instrument, such as the communication protocols to use. You must consider the following issues with PC control of instrumentation:

- Type of connector (pinouts) on the instrument
- Cables needed—null-modem, number of pins, male/female
- Electrical properties involved—signal levels, grounding, cable length restrictions
- Communication protocols used—ASCII commands, binary commands, data format
- Software drivers available

## 10.2 GPIB COMMUNICATION

The ANSI/IEEE Standard 488.1-1987, also known as General Purpose Interface Bus (GPIB), describes a standard interface for communication between instruments and controllers from various

vendors. GPIB instruments offer test and manufacturing engineers the widest selection of vendors and instruments for general-purpose to specialized vertical market test applications as shown in Figure 10.3. GPIB instruments are often used as stand-alone benchtop instruments where measurements are taken by hand. You can automate these measurements by using a PC to control the GPIB instruments.



**Figure 10.3** GPIB communication.

IEEE 488.1 contains information about electrical, mechanical and functional specifications. The ANSI/IEEE Standard 488.2-1992 extends IEEE 488.1 by defining a bus communication protocol, a common set of data codes and formats, and a generic set of common device commands. GPIB is a digital, 8-bit parallel communication interface with data transfer rates of 1 Mbyte/s and higher, using a three-wire handshake. The bus supports one system controller, usually a computer, and up to 14 additional instruments. The GPIB protocol categorizes devices as controllers, talkers, or listeners to determine which device has active control of the bus. Each device has a unique GPIB primary address between 0 and 30. The controller defines the communication links, responds to devices that request service, sends GPIB commands, and passes/receives control of the bus. Controllers instruct talkers to talk and to place data on the GPIB. You can address only one device at a time to talk. The controller addresses the listener to listen and to read data from the GPIB. You can address several devices to listen.

### 10.2.1 Data Transfer Termination

Termination informs listeners that all data has been transferred. You can terminate a GPIB data transfer in the following three ways:

- The GPIB includes an end-of-Identify (EOI) hardware line that can be asserted with the last data byte. This is the preferred method.
- Place a specific end-of-string (EOS) character at the end of the data string itself. Some instruments use this method instead of or in addition to the EOI line assertion.

- The listener counts the bytes transferred by handshaking and stops reading when the listener reaches a byte count limit. This method is often used as a default termination method because the transfer stops on the logical OR of EOI, EOS (if used) in conjunction with the byte count. Thus, you typically set the byte count to equal or exceed the expected number of bytes to be read.

### 10.2.2 Data Transfer Rate

To achieve the high data transfer rate that the GPIB was designed for, you must limit the number of devices on the bus and the physical distance between devices. You can obtain faster data rates with HS488 devices and controllers. HS488 is an extension to GPIB that most NI controllers support.

## 10.3 HARDWARE SPECIFICATIONS

The GPIB is a digital, 24-conductor parallel bus. As shown in Figure 10.4, it consists of eight data lines (DIO 1-8), five bus management lines (EOI, IFC, SRQ, ATN, REN), three handshake lines (DAV, NRFD, NDAC), and eight ground lines. The GPIB uses an eight-bit parallel, byte-serial, asynchronous data transfer scheme. This means that whole bytes are sequentially handshaked across the bus at a speed that the slowest participant in the transfer determines. Because the unit of data on the GPIB is a byte (eight bits), the messages transferred are frequently encoded as ASCII character strings.

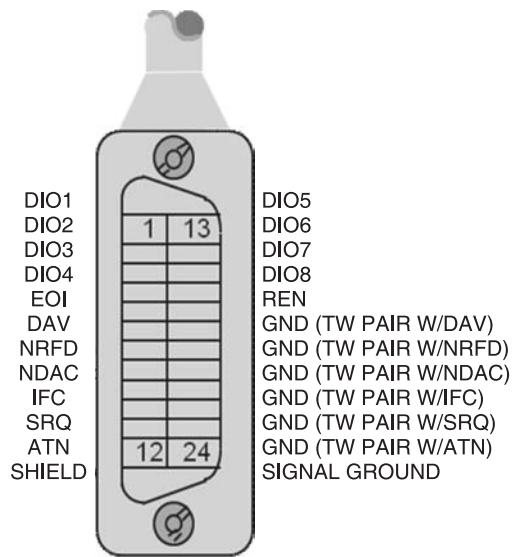


Figure 10.4 Hardware specification.

Additional electrical specifications allow data to be transferred across the GPIB at the maximum rate of 1 MB/sec because the GPIB is a transmission line system. These specifications are:

- A maximum separation of 4 m between any two devices and an average separation of 2 m over the entire bus.

- A maximum cable length of 20 m.
- A maximum of 15 devices connected to each bus with at least two-thirds of the devices powered on.

If you exceed any of these limits, you can use additional hardware to extend the bus cable lengths or expand the number of devices allowed.

#### 10.4 SOFTWARE ARCHITECTURE

The software architecture for the instrument control using LabVIEW is similar to the architecture for DAQ. Instrument interfaces such as GPIB include a set of drivers. Use MAX to configure the interface. VISA, Virtual Instrument Software Architecture, is a common API to communicate with the interface drivers and is the preferred method used when programming for instrument control in LabVIEW, because VISA abstracts the type of interface used. Many LabVIEW VIs used for instrument control use the VISA API. For example, the Instrument I/O Assistant is a LabVIEW Express VI that can use VISA to communicate with message-based instruments and convert the response from raw data to an ASCII representation. Use the Instrument I/O Assistant when an instrument driver is not available. In LabVIEW, an instrument driver is a set of VIs specially written to communicate with an instrument.

Underneath and between the different hardware components are the software components that are the heart of your instrumentation system. Figure 10.5 shows the software architecture for Windows 2000/XP from the perspective of the computer. The instrument is connected to the computer through a built-in connector such as RS-232 or an installed interface board such as GPIB or VXI/MXI. Driver-level software for instrument is in the form of a Dynamic Link Library (DLL). In the example shown in the figure, LabVIEW is the high-level application software layer. LabVIEW includes built-in functions for GPIB, serial, VXI, and computer-based instruments that load and use the installed driver software.

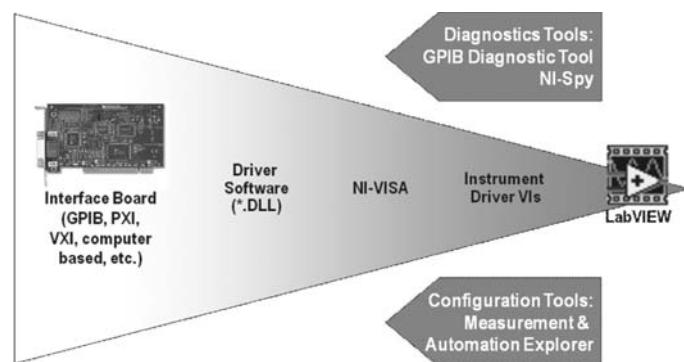
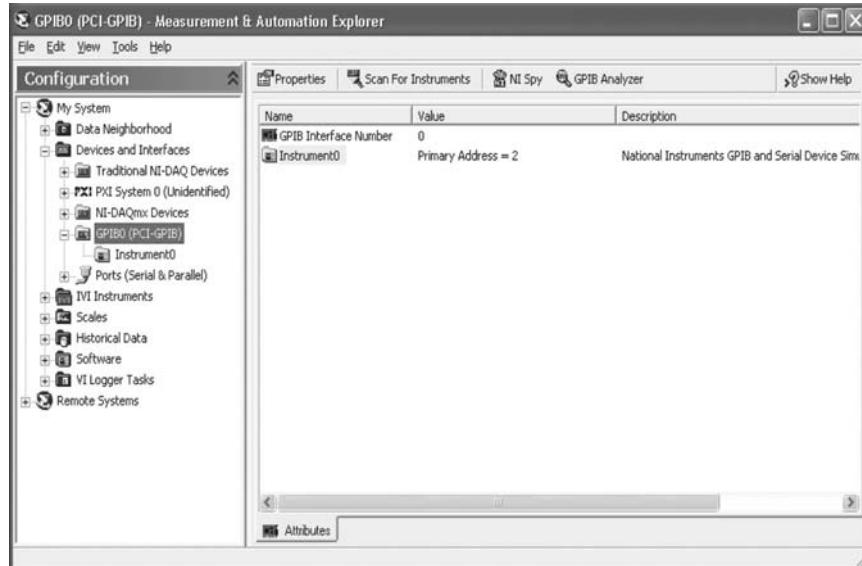


Figure 10.5 Software architecture for Windows.

As with all Windows device drivers, any DLLs you install for interface boards will also interact with the Windows Registry so that resources such as base addresses, interrupt levels and DMA channels can be assigned. Configuration and diagnostic software tools are also available with National Instruments hardware.

### 10.4.1 MAX (Windows; GPIB)

Use MAX to configure and test the GPIB interface. MAX interacts with the various diagnostic and configuration tools installed with the driver and also with the Windows Registry and Device Manager. The driver-level software is in the form of a DLL and contains all the functions that directly communicate with the GPIB interface. The Instrument I/O VIs and functions directly call the driver software. Open MAX by double-clicking the icon on the desktop or by selecting *Tools»Measurement & Automation Explorer* in LabVIEW to open the window as in Figure 10.6.



**Figure 10.6** GPIB Interface in Measurement and Automation Explorer.

Configure the objects listed in MAX by right-clicking each item and selecting an option from the shortcut menu. The Measurement and Automation Explorer (MAX) allows you to configure the GPIB card and search for instruments, set the VISA alias name, run NI-Spy and configure IVI instrument drivers. If a GPIB+ card is installed, you can run the GPIB analyzer software from MAX.

## 10.5 INSTRUMENT I/O ASSISTANT

The Instrument I/O Assistant is a LabVIEW Express VI which you can use to communicate with message-based instruments and convert the response from raw data to an ASCII representation. You can communicate with an instrument that uses a serial, Ethernet, or GPIB interface. The Instrument I/O Assistant organizes instrument communication into ordered steps. To use Instrument I/O Assistant, you place steps into a sequence. As you add steps to the sequence, they appear in the Step Sequence window. Use the view associated with a step to configure instrument I/O.

The Instrument I/O Assistant shown in Figure 10.7 is located on the *Functions»Input* and *Functions»All Functions»Instrument I/O* palettes is a LabVIEW Express VI that allows you to

easily test communication with your instrument and develop a sequence of query, parse and write steps. These steps can be saved as an Express VI for instant use or can be converted to a LabVIEW subVI. Use the Instrument I/O Assistant when an instrument driver is not available.



**Figure 10.7** Instrument I/O Assistant.

To launch the Instrument I/O Assistant, place the Instrument I/O Assistant Express VI on the block diagram in LabVIEW. The Instrument I/O Assistant Express VI is available in the Instrument I/O category of the *Functions* palette. The *Instrument I/O Assistant* configuration dialog box appears. If it does not appear, double-click the Instrument I/O Assistant icon. Complete the following steps to configure the Instrument I/O Assistant.

**Step 1:** Select an instrument. Instruments that have been configured in MAX appear in the *Select an instrument* pull-down menu.

**Step 2:** Choose a *Code generation type*. VISA code generation allows for more flexibility and modularity than GPIB code generation.

**Step 3:** Select from the following communication steps using the *Add Step* button:

- **Query and Parse**—Sends a query to the instrument, such as \*IDN? and parses the returned string. This step combines the Write command and Read and Parse command.
- **Write**—Sends a command to the instrument.
- **Read and Parse**—Reads and parses data from the instrument.

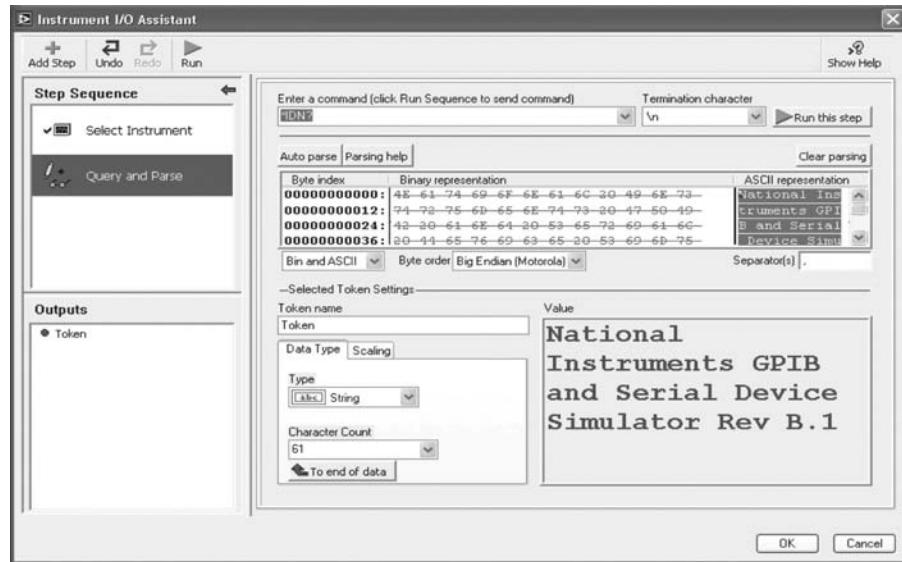
**Step 4:** After adding the desired number of steps, click the *Run* button to test the sequence of communication that you have configured for the Express VI.

**Step 5:** Click the *OK* button to exit the *Instrument I/O Assistant* configuration dialog box.

LabVIEW adds input and output terminals to the Instrument I/O Assistant Express VI on the block diagram that correspond to the data you receive from the instrument. To view the code generated by the Instrument I/O Assistant, right-click the Instrument I/O Assistant icon and select *Open Front Panel* from the shortcut menu. This converts the Express VI to a subVI. Switch to the block diagram to see the code generated. After you convert an Express VI to a subVI, you cannot reconvert the Express VI.

Once you have placed the I/O Assistant on the block diagram, the wizard opens. The wizard starts in the Select Instrument step, where you can choose a GPIB or serial instrument. You also can check the interface properties from this window. After selecting the instrument, you can add sequences to Query and Parse, Write, or Read and Parse. In addition, after you have set up a

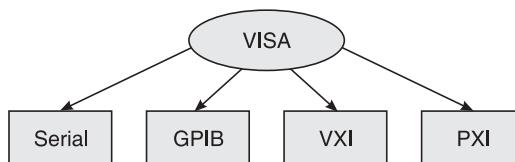
communication sequence with one instrument, you can set up additional instruments in the same Express VI. In Figure 10.8, a GPIB instrument was set up, then a query (\*IDN?) was sent to the instrument. The response was automatically parsed, resulting in a string output.



**Figure 10.8** Serial Configuration of the Instrument I/O Assistant.

## 10.6 VISA

Virtual Instrument Software Architecture (VISA) is the lower layer of functions in the LabVIEW instrument driver VIs that communicates with the driver software. VISA by itself does not provide instrumentation programming capability. VISA is a high-level API that calls low-level drivers. As shown in Figure 10.9 VISA can control VXI, GPIB, serial, or computer-based instruments and makes the appropriate driver calls depending on the type of instrument used. When debugging VISA problems, remember that an apparent VISA problem could be an installation problem with one of the drivers that VISA calls.



**Figure 10.9** Virtual Instrument Software Architecture.

In LabVIEW, VISA is a single library of functions you use to communicate with GPIB, serial, VXI and computer-based instruments. You do not need to use separate I/O palettes to program an instrument. For example, some instruments give you a choice for the type of interface. If the LabVIEW instrument driver was written with functions on the *Functions»All Functions»Instrument*

*I/O»GPIB* palette, those instrument driver VIs would not work for the instrument with the serial port interface. VISA solves this problem by providing a single set of functions that work for any type of interface. Therefore, many LabVIEW instrument drivers use VISA as the I/O language.

For many years, industry has moved toward purchasing instrumentation from a variety of vendors. This allows engineers to select the best possible equipment for their applications without being locked into a specific vendor. This trend required the definition of hardware standards to ensure the compatibility between different modules. This was one of the factors leading to the development of the VXI specification. But even with these improved hardware standards, a system was time consuming and expensive to put together. Successful integration of a multivendor system requires all hardware and software products work together, eliminating system-level compatibility issues for end-users. National Instruments initially addressed the software problems with instrument drivers, which helped reduce both integration time and software development costs. In 1993, National Instruments joined with GenRad, Racal Instruments, Tektronix and Wavetek to form the VXI plug & play Systems Alliance. The goals of the alliance are to ensure multivendor interoperability for VXI systems and to reduce the development time for an operational system.

### 10.6.1 VISA Programming Terminology

VISA or Virtual Instrument Software Architecture is a protocol built upon 488.2 driver and functions to meet the industry needs for having a way to easily interface with multiple I/Os and have all manufacturers of instruments and instrument drivers follow a protocol. VISA created by the VXIplug&play Alliance which is composed of the top 35 instrument manufacturers such as HP. National Instruments is a leading member of the alliance. The resource name contains information on the type of I/O interface and the device address. You can use an alias you assign in MAX instead of the instrument descriptor. (Mac OS) Edit the visaconf.ini file to assign a VISA alias. (UNIX) Use the visaconf utility. If you choose not to use the Instrument I/O Assistant to automatically generate code for you, you can still write a VI to communicate with the instrument.

Before being introduced to VISA programming, you should become familiar with some of the VISA terminology. The most important objects in the VISA language are known as resources. The functions you can use with an object are known as operations. In addition to the operations that you can use an object, the object has variables, known as attributes, associated with it that contains information related to the object. Three terms need definitions and the following terminology is similar to that used for instrument driver VIs.

- **Resource**—Any instrument in the system, including serial and parallel ports.
- **Session**—You must open a VISA session to a resource to communicate with it, similar to a communication channel. When you open a session to a resource, LabVIEW returns a VISA session number which is a unique refnum to that instrument. You must use the session number in all subsequent VISA functions.
- **Instrument Descriptor**—Exact name of a resource. The descriptor specifies the interface type (GPIB, VXI, ASRL), the address of the device (logical address or primary address) and the VISA session type (INSTR or Event). The instrument descriptor is similar to a telephone number, the resource is similar to the person with whom you want to speak and the session is similar to the telephone line. Each call uses its own line, and crossing these lines results in an error. Table 10.1 shows the proper syntax for the instrument descriptor.

**TABLE 10.1** Syntax for various instrument interfaces

Interface	Resource Name Grammar
Serial	ASRL[board][::INSTR]
GPIB	GPIB[board]::primary address[::INSTR]
VXI	VXI[board]::VXI logical address[::INSTR]
GPIB-VXI	GPIB-VXI[board]::GPIB-VXI primary address ::VXI logical address[::INSTR]

The most commonly used VISA communication functions are the VISA Write and VISA Read functions. Most instruments require you to send information in the form of a command or query before you can read information back from the instrument. Therefore, the VISA Write function is usually followed by a VISA Read function. The VISA Write and VISA Read functions work with any type of instrument communication and are the same whether you are doing GPIB or serial communication. However, because serial communication requires you to configure extra parameters, you must start the serial port communication with the VISA Configure Serial Port VI.

### 10.6.2 VISA and Serial

The VISA Configure Serial Port VI initializes the port identified by *VISA resource name* to the specified settings. *Timeout* sets the timeout value for the serial communication. *Baud rate*, *data bits*, *parity* and *flow control* specify those specific serial port parameters. The *error in* and *error out* clusters maintain the error conditions for this VI.

The VISA Configure Serial Port VI opens communication with COM2 and sets it to 9,600 baud, eight data bits, odd parity, one stop bit and XON/XOFF software handshaking. Then, the VISA Write function sends the command. The VISA Read function reads back up to 200 bytes into the read buffer, and the Simple Error Handler VI checks the error condition. The VIs and functions located on the *Functions»All Functions»Instrument I/O»Serial* palette are also used for parallel port communication. You specify the VISA resource name as being one of the LPT ports.

## 10.7 INSTRUMENT DRIVERS

LabVIEW provides more than 1200 LabVIEW instrument drivers from more than 50 vendors. You can use these instrument drivers to build complete systems quickly. Instrument drivers drastically reduce software development costs because developers do not need to spend time programming their instruments. You can reuse the drivers in a variety of systems and configurations. LabVIEW instrument drivers simplify instrument programming to high-level commands, so you do not need to learn the low-level instrument-specific syntax needed to control your instruments. The programming is broken down into general functions for the DMM. These instrument drivers are called LabVIEW instrument drivers because the source code is graphical programming made from standard LabVIEW functions and VIs. LabVIEW instrument drivers are in the form of VI libraries and are organized into categories of instrument functions.

Consider an example where you wrote a LabVIEW VI that communicates with a specific oscilloscope in your lab. Unfortunately, the oscilloscope no longer works, and you must replace it. However, this particular oscilloscope is no longer made. You found a different brand of oscilloscope that you want to purchase, but your VI no longer works with the new oscilloscope. You must rewrite your VI. When you use an instrument driver, the driver contains the code specific to the instrument. Therefore, if you change instruments, you must replace only the instrument driver VIs with the instrument driver VIs for the new instrument, which greatly reduces your redevelopment time. Instrument drivers help make test applications easier to maintain because the drivers contain all the I/O for an instrument in one library, separate from other code. When you upgrade hardware, upgrading the application is easier because the instrument driver contains all the code specific to that instrument.

A LabVIEW Plug and Play instrument driver is a set of VIs that control a programmable instrument. Each VI corresponds to an instrument operation, such as configuring, triggering and reading measurements from the instrument. Instrument drivers help users get started using instruments from a PC and saves them development time and cost because users do not need to learn the programming protocol for each instrument. With open-source, well documented instrument drivers, end users can customize their operation for better performance. A modular design makes the driver easier to customize.

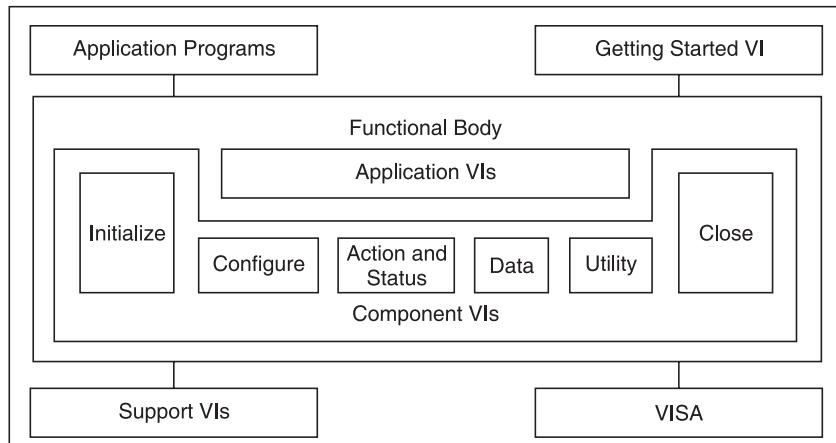
You can locate most LabVIEW Plug and Play instrument driver in the Instrument Driver Finder. You can access the Instrument Driver Finder within LabVIEW by selecting *Tools»Instrumentation»Find Instrument Drivers* or *Help»Find Instrument Drivers*. The Instrument Driver Finder connects you with ni.com to find instrument drivers. You can find the LabVIEW instrument drivers on your LabVIEW installation CD or download them from the National Instruments web page at [www.ni.com/idnet](http://www.ni.com/idnet). The Instrument Driver Network at ni.com is the industry's largest source of instrument drivers, featuring drivers for over 2,200 instruments from 150 vendors. Here, you can learn about, download, develop, and submit drivers for controlling instruments from LabVIEW, LabWindows/CVI, and Measurement Studio. The LabVIEW Instrument Wizard can automatically install instrument drivers. If you install the LabVIEW instrument drivers from the CD yourself or download them from the web page, you first decompress the instrument driver files to get a directory of instrument driver files. Place this directory into the LabVIEW\instr.lib directory on your computer. The next time you launch LabVIEW, you can access the instrument driver VIs from the *Input»Instrument Drivers* subpalette of the *Functions* palette.

Many programmable instruments have a large number of functions and modes. With this complexity, it is necessary to provide a consistent design model that aids both instrument driver developers as well as end users who develop instrument control applications. The LabVIEW Plug and Play instrument driver model contains both external structure and internal structure guidelines. The external structure shows how the instrument driver interfaces with the user and to other software components in the system. The internal structure shows the internal organization of the instrument driver software module. For the external structure of the instrument driver, the user interacts with the instrument driver using an API or an interactive interface. Usually, the interactive interface is used for testing or for end-users. The API is accessed through LabVIEW. The instrument driver communicates with the instrument using VISA. Internally, the VIs in an instrument driver are organized into six categories. These categories are summarized in Table 10.2.

**TABLE 10.2** VIs in an instrument driver

Category	Description
Initialize	The initialize VI establishes communication with the instrument and is the first instrument driver VI called.
Configure	This collection of VIs are software routines that configure the instrument to perform specific operations. After calling these VIs, the instrument is ready to take measurements or stimulate a system.
Action/Status	This collection of VIs command the instrument to carry out an action (i.e. arming a trigger) or obtain the current status of the instrument or pending operations.
Data	The data VIs transfer data to or from the instrument.
Utility	This collection of VIs perform a variety of auxiliary operations, such as reset and self-test.
Close	The close VI terminates the software connection to the instrument. This is the last instrument driver VI called.

All instrument drivers in the library have the same VI Tree structure. Therefore, once you learn to use one instrument driver, all others have the same basic hierarchy. In fact, this hierarchy, sequence of VIs and error checking are used in many other areas of I/O in LabVIEW such as file I/O, data acquisition (DAQ) and TCP/IP communications.

**Figure 10.10** Instrument driver model.

The structure of an instrument driver is shown in Figure 10.10. The high-level functions are built from the lower-level functions. For the most control over the instrument, you would use the lower-level functions. However, the high-level functions like the *Getting Started VI* you used in the previous lesson are easy to use and have soft front panels that resemble the instrument.

### 10.7.1 Instrument Driver VIs

As shown in Figure 10.11 an instrument driver VI initializes the DMM with its VISA Alias, uses an Application Example VI to configure and read data from the meter, and closes the instrument, and then the error status is checked. You will see this same sequence of events in every application that

uses an instrument driver. Notice how the Instrument Descriptor, VISA Sessions, and Error I/O terminals are wired. Remember that you can right click on the instrument driver VI terminals and choose Create Constant, Create Control, or Create Indicator as needed. Instrument drivers drastically reduce software development costs because developers do not need to spend time programming their instruments. You can reuse the drivers in a variety of systems and configurations. The instrument driver VIs are as follows:

**Initialize**—Initializes the communication channel to the instrument. The initialize VI can optionally perform an identification query and reset operation. In addition, it can perform any necessary actions to place the instrument in its default power-on state or other specified state.

**Configure**—A collection of VIs that configure the instrument for the operations you want to perform. An example is a function to set up the trigger rate.

**Action/Status**—Contains both action and status VIs. Action VIs cause the instrument to initiate or terminate test and measurement operations. Status VIs obtain the current status of the instrument or the status of pending operations. An example of an action function is *Acquire Single Shot*. An example of a status function is *Query Transfer Pending*.

**Data**—VIs that transfer data to or from the instrument such as reading a measured waveform from the instrument or downloading a waveform to the instrument.

**Utility**—VIs that perform a wide variety of useful functions such as reset, self-test, error query and revision query.

**Close**—Terminates the communication channel to the instrument and de-allocates the resources set aside for that instrument.

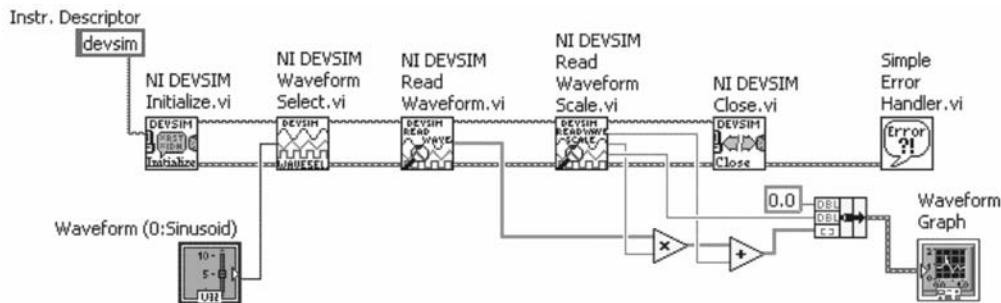


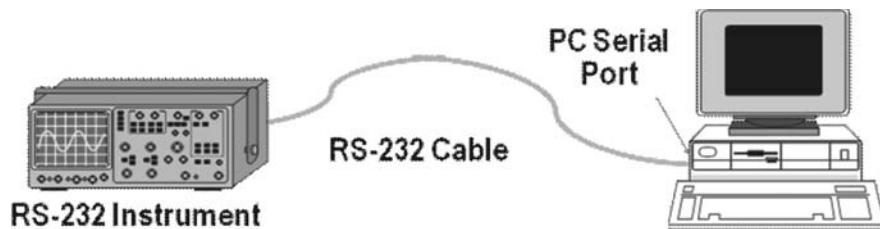
Figure 10.11 VIs to communicate with an instrument.

LabVIEW instrument drivers simplify instrument programming to high-level commands, so you do not need to learn the low-level instrument-specific syntax needed to control your instruments. The programming is broken down into general functions for the DMM.

## 10.8 SERIAL PORT COMMUNICATIONS

Serial communication is a popular means of transmitting data between a computer and a peripheral device such as a programmable instrument or even another computer. Serial communication uses

a transmitter to send data, one bit at a time, over a single communication line to a receiver. You can use this method when data transfer rates are low or you must transfer data over long distances. Serial communication is popular because most computers have one or more serial ports, so no extra hardware is needed other than a cable to connect your instrument to the computer (or two computers together) as shown in Figure 10.12.

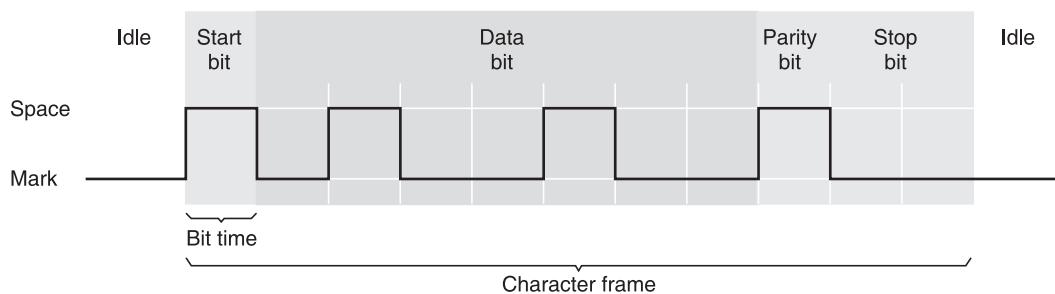


**Figure 10.12** Serial instrument.

You must specify four parameters for serial communication: the baud rate of the transmission, the number of data bits that encode a character, the sense of the optional parity bit, and the number of stop bits. A character frame packages each transmitted character as a single start bit followed by the data bits. The baud rate is a measure of how fast data moves between instruments that use serial communication. The baud rate informs how many bits are transferred per second on the serial cable.

Data bits indicate how many bits represent a data value. Data bits are transmitted upside down and backwards, which means that inverted logic is used and the order of transmission is from the least significant bit (LSB) to the most significant bit (MSB). Parity provides optional error checking bit that is added to the data. Stop bits are a certain number of bits added to the end of each data transfer. Flow control is optional hardware or software handshaking parameters for communicating with a device.

To interpret the data bits in a character frame as shown in Figure 10.13, you must read from right to left and read 1 for a negative voltage and 0 for a positive voltage. An optional parity bit follows the data bits in the character frame. The parity bit, if present, also follows inverted logic. This bit is included as a means of error checking. You specify ahead of time for the parity of the transmission to be even or odd. If you choose for the parity to be odd, the parity bit is set in such a way so the number of 1s add up to make an odd number among the data bits and the parity bit. The



**Figure 10.13** A typical serial communication.

last part of a character frame consists of 1, 1.5, or 2 stop bits that are always represented by a negative voltage. If no further characters are transmitted, the line stays in the negative (MARK) condition. The transmission of the next character frame, if any, begins with a start bit of positive (SPACE) voltage.

RS-232 uses only two voltage states called MARK and SPACE. In such a two-state coding scheme, the baud rate is identical to the maximum number of bits of information, including control bits that are transmitted per second. MARK is a negative voltage, and SPACE is positive. The following is the truth table for RS-232:

Signal  $> +3 \text{ V} = 0$   
 Signal  $< -3 \text{ V} = 1$

The output signal level usually swings between  $+12 \text{ V}$  and  $-12 \text{ V}$ . The dead area between  $+3 \text{ V}$  and  $-3 \text{ V}$  is designed to absorb line noise. A start bit signals the beginning of each character frame. It is a transition from negative (MARK) to positive (SPACE) voltage. Its duration in seconds is the reciprocal of the baud rate. If the instrument is transmitting at 9,600 baud, the duration of the start bit and each subsequent bit is about 0.104 ms. The entire character frame of eleven bits would be transmitted in about 1.146 ms. Interpreting the data bits for the transmission yields 1101101 (binary) or 6D (hex). An ASCII conversion table shows that this is the letter m. This transmission uses odd parity. There are five ones among the data bits, already an odd number, so the parity bit is set to 0. Select COMX as the instrument address. Use the I/O Assistant just like with GPIB communication as shown in Figure 10.14.

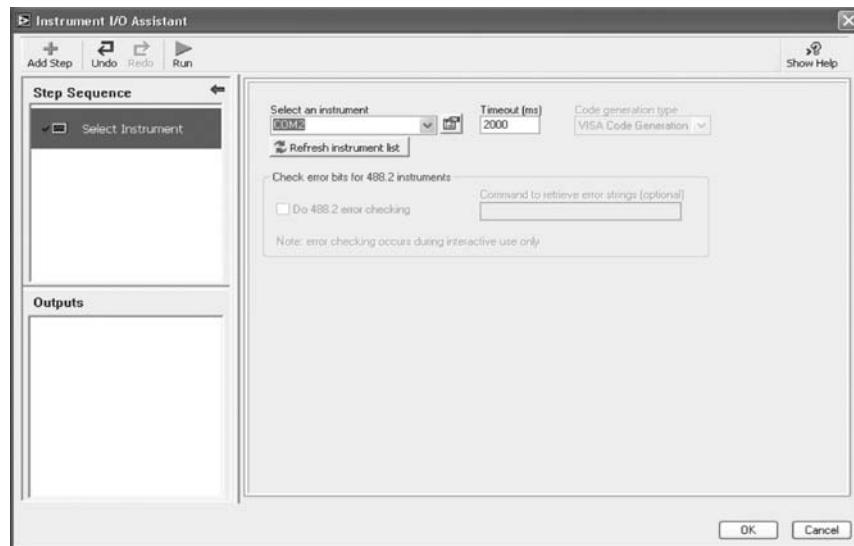


Figure 10.14 Instrument I/O Assistant with serial communication.

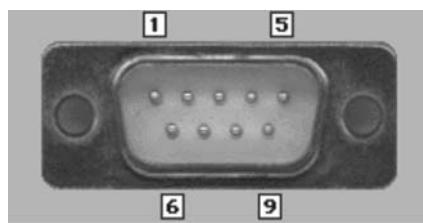
### 10.8.1 Data Transfer Rate

You can calculate the maximum transmission rate in characters per second for a given communication setting by dividing the baud rate by the bits per character frame. In the previous

example, there are a total of eleven bits per character frame. If the transmission rate is set at 9,600 baud, you get  $9,600 / 11 = 872$  characters per second. Notice that this is the maximum character transmission rate. The hardware on one end or the other of the serial link might not be able to reach these rates, for various reasons.

### 10.8.2 Serial Port Standards

There are several recommended standards (RS) for serial communication. Each varies in hardware and software specifications. One must be familiar with their instrument and what connector is used before you can begin controlling that device with their computer. Figure 10.15 shows serial hardware connection.



**Figure 10.15** Serial hardware connections.

There are three main Serial I/O types that are most common recommended standards of serial port communication.

- RS-232 (ANSI/EIA-232 Standard) is used for many purposes such as connecting a mouse, printer, or modem. It is also used with industrial instrumentation. Because of improvements in line drivers and cables, applications often increase the performance of RS-232 beyond the distance and speed in the standards list. RS-232 is limited to point-to-point connections between PC serial ports and devices. RS-232 is the connector found on most PCs. This is a single-ended communication method where only one device can be connected per port. Two connector types are 9 or 25-pin. Two configurations are DCE or DTE.
- RS-422 (AIA RS-422A Standard) uses a differential electrical signal as opposed to the unbalanced (single-ended) signals referenced to ground with RS-232. Differential transmission, which uses two lines each to transmit and receive signals, results in greater noise immunity and longer transmission distances as compared to RS-232. RS-422 is the Connector found on most Macs. This is a differential communication method. Connector has 8 pins.
- RS-485 (EIA-485 Standard) is a variation of RS-422 that allows you to connect up to 32 devices to a single port and define the necessary electrical characteristics to ensure adequate signal voltages under maximum load. With this enhanced multidrop capability, you can create networks of devices connected to a single RS-485 serial port. The noise immunity and multidrop capability make RS-485 an attractive choice in industrial applications that require many distributed devices networked to a PC or other controller for data collection and other operations.

## 10.9 USING OTHER INTERFACES

There are devices made to communicate with serial or GPIB instruments through the Ethernet, USB, or IEEE 1394 (FireWire) ports which bypasses the need for a serial port or GPIB board on your computer. When using these devices, program them just as you would if they were using the serial port or a GPIB board. USB and ethernet interfaces transform USB ports or ethernet ports into asynchronous serial ports for communication with serial instruments. You can install and use these interfaces as standard serial ports from your existing applications. USB, ethernet, and IEEE 1394 controllers transform any computer with these ports into a full-function, Plug and Play, IEEE-488.2 Controller that can control up to 14 programmable GPIB instruments.

## SUMMARY

---

- LabVIEW can communicate with any instrument that connects to your computer if you know the interface type.
- Use the Measurement & Automation Explorer (MAX) to detect, configure and test your GPIB interface and instruments.
- Use the Instrument I/O Assistant for easy and fast GPIB and serial programming.
- Maximum cable length between GPIB devices = 4 m (2 m average)
- Maximum cable length = 20 m
- Maximum number of devices = 15 (2/3 powered on)
- An instrument driver eliminates the need to know low level commands. It is a collection of VIs used to control an instrument.
- Instrument driver VIs share a common hierarchy and come with an example to help you get started.
- Serial communication is a popular means of communication between computer and peripheral device.
- In serial communication data sent is one bit at a time across the cable. It is used for low transfer rates or long distances.
- Most computers have at least one available serial port.
- *Baud rate*—bits per second
- *Data bits*—inverted logic and LSB first
- *Parity*—optional error-checking bit
- *Stop bits*—1, 1.5, or 2 inverted bits at data end
- *Flow control*—hardware and software handshaking options
- *RS-232*—connector found on most PCs.

## MISCELLANEOUS SOLVED PROBLEMS

---

**Problem 10.1** Use MAX to examine the GPIB interface settings, detect instruments, and communicate with an instrument.

**Solution** The following steps help understand GPIB configuration with MAX (Windows Only).

**Step 1:** Power off the NI Instrument Simulator and configure it to communicate through GPIB by setting the left bank of switches on the side of the box [Figure P10.1(a)].

**Step 2:** Power on the NI Instrument Simulator and verify that both the Power and Ready LEDs are lit.

**Step 3:** Launch MAX by either double-clicking the icon on the desktop or by selecting *Tools»Measurement & Automation Explorer* in LabVIEW.

**Step 4:** Expand the *Devices and Interfaces* section to display the installed interfaces. If a GPIB interface is listed, the NI-488.2 software is correctly loaded on the computer.

**Step 5:** Select the GPIB interface and click the *Properties* button on the toolbar to display the *Properties* dialog box.

**Step 6:** Examine but do not change the settings for the GPIB interface and click the *OK* button.

**Step 7:** Make sure the GPIB interface is still selected in the *Devices and Interfaces* section and click the *Scan for Instruments* button on the toolbar.

**Step 8:** Expand the GPIB board section. One instrument named Instrument 0 appears.

**Step 9:** Click *Instrument 0* to display information about it in the right pane of MAX. The NI Instrument Simulator has a GPIB primary address (PAD) of 2.

**Step 10:** Click the *Communicate with Instrument* button on the toolbar. An interactive window appears. You can use it to query, write to, and read from that instrument.

**Step 11:** Type \*IDN? in *Send String* and click the *Query* button [Figure P10.1(b)]. The instrument returns its make and model number in *String Received*. You can use this window to debug instrument problems or to verify that specific commands work as described in the instrument documentation.

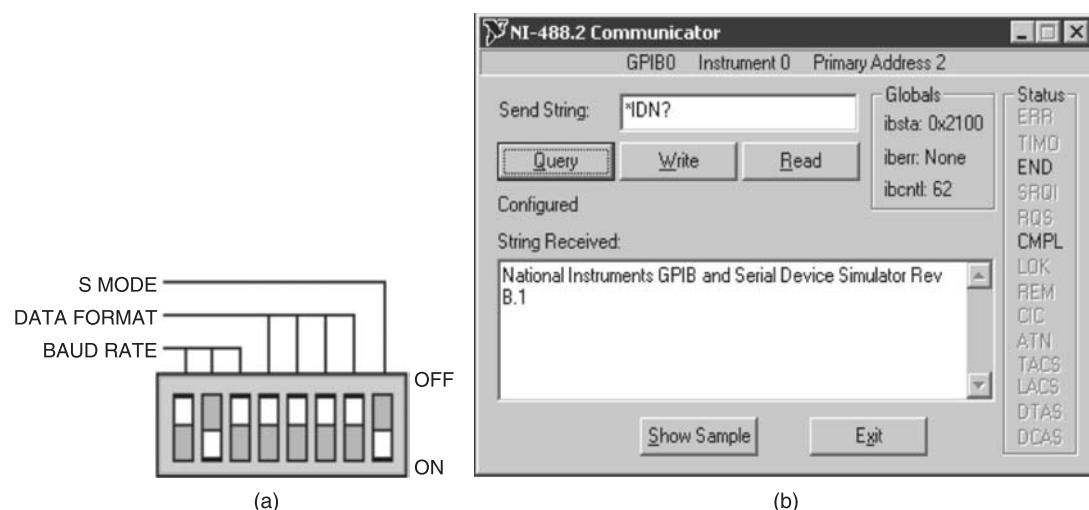


Figure P10.1

**Step 12:** Type MEAS:DC? in *Send String* and click the *Query* button. The NI Instrument Simulator returns a simulated voltage measurement.

**Step 13:** Click the *Query* button again to return a different value.

**Step 14:** Click the *Exit* button.

**Step 15:** Set a VISA alias for the NI Instrument Simulator, so you can use the alias instead of having to remember the primary address.

- (a) While *Instrument0* is selected in MAX, click the *VISA Properties* button to display the *Properties* dialog box.
- (b) Type devsim in the *VISA Alias* field and click the *OK* button. You will use this alias throughout this lesson.

**Step 16:** Select *File»Exit* to exit MAX.

**Problem 10.2** Build a VI that uses the Instrument I/O Assistant to communicate with a GPIB interface.

**Solution** Complete the following steps to build a VI that acquires data from the NI Instrument Simulator using the Instrument I/O Assistant and the solution steps to the problem are given below.

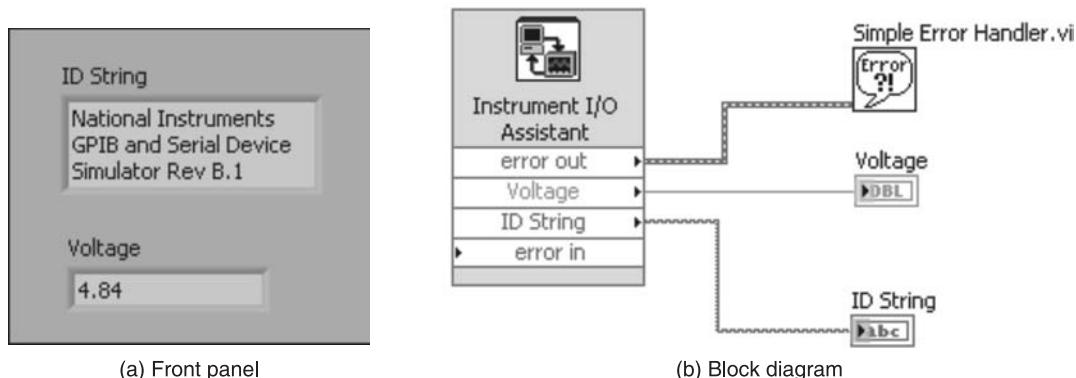
### Front Panel

**Step 1:** Open a blank VI.

**Step 2:** The front panel as shown in Figure P10.2(a) will result from building the block diagram.

### Block Diagram

**Step 3:** Display and build the block diagram as shown in Figure P10.2(b).



**Figure P10.2**



- (a) Place the Instrument I/O Assistant Express VI, located on the *Functions» Input* palette, on the block diagram. Complete the following steps to configure the Express VI in the *Instrument I/O Assistant* dialog box.

- (i) Select *devsim* from the *Select an instrument* pull-down menu and select *VISA Code Generation* from the *Code generation type* pull-down menu.
- (ii) Click the *Add Step* button. Click *Query and Parse* to write and read from the Instrument Simulator.
- (iii) Type *\*IDN?* as the command, select *\n* as the *Termination character*, and click the *Run this step* button. If no error warning appears in the lower half of the dialog box, this step has successfully completed.
- (iv) To parse the data received, click the *Auto parse* button. Notice that *Token* now appears in the *Outputs* pane on the left side of the dialog box. This value represents the string returned from the identification query. Rename *Token* by typing *ID String* in the *Token name* text box.
- (v) Click the *Add Step* button. Click *Query and Parse*. Type *MEAS:DC?* as the command and click the *Run this step* button.
- (vi) To parse the data received, click the *Auto parse* button. The data returned is a random numeric value. Rename *Token* by typing *Voltage* in the *Token name* text box.
- (vii) Click the *OK* button to exit the I/O Assistant and return to the block diagram.
- (b) Right-click the *ID String* output and select *Create» Indicator* from the shortcut menu.
- (c) Right-click the *Voltage* output and select *Create» Indicator* from the shortcut menu.
- (d) Wire the *Error Out* output to the Simple Error Handler VI.

**Step 4:** Display the front panel and run the VI. Resize the string indicator if necessary.

**Step 5:** Save the VI as Read Instrument Data.vi in the C:\Exercises\LabVIEW Basics I directory.

**Step 6:** Right-click the I/O Assistant and select *Open Front Panel*. Click the *Convert* button when asked if you want to convert to a subVI.

**Step 7:** View the code generated by the I/O Assistant. Where is the command *\*IDN?* written to the Instrument Simulator? Where is the voltage being read?

**Step 8:** Select *File» Exit* to exit the sub VI. Do not save changes.

**Problem 10.3** Build a VI that reads and writes information from the NI Instrument Simulator using VISA functions.

**Solution** Complete the following steps to build a solution VI for the problem that uses VISA calls to acquire data from the NI Instrument Simulator and understand programming with VISA.

**Step 1:** Make sure the Instrument Simulator is powered on and connected to the GPIB Interface.

#### Front Panel

**Step 2:** Open a blank VI. The front panel shown in Figure P10.3(a) will result from building the block diagram.

#### Block Diagram

**Step 3:** Build the block diagram as shown in Figure P10.3(b).

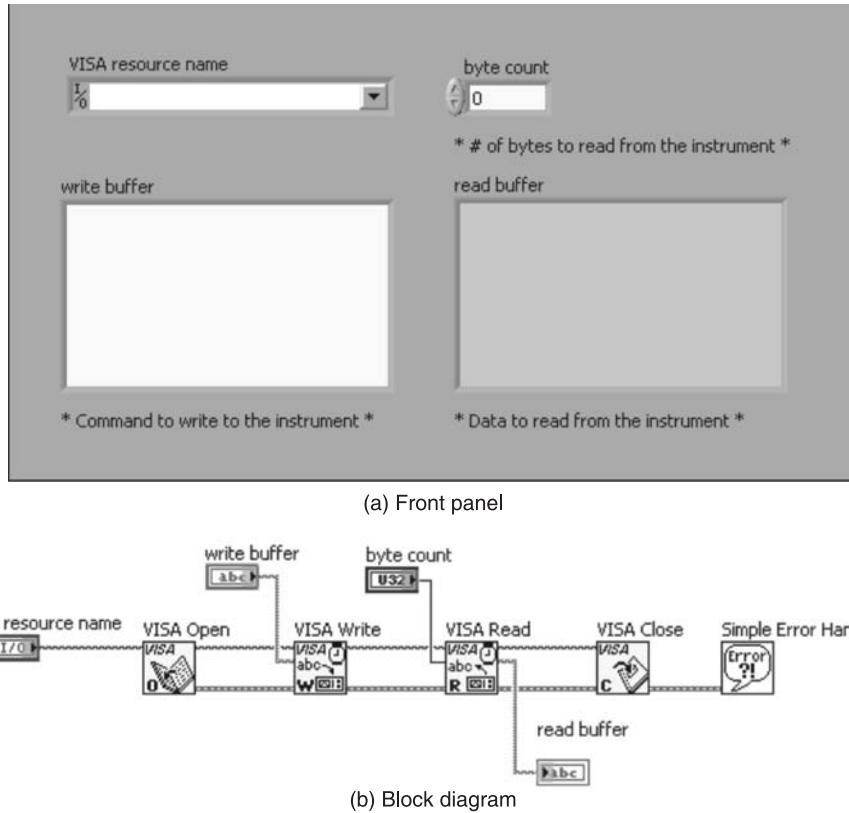


Figure P10.3



(a) Place the VISA Open function, located on the *Functions» All Functions» Instrument I/O»VISA»VISA Advanced* palette, on the block diagram. This function opens a VISA session with an instrument. Right-click the *VISA resource name* input and select *Create» Control* from the shortcut menu.



(b) Place the VISA Write function, located on the *Functions» All Functions» Instrument I/O»VISA* palette, on the block diagram.

This function writes a string to the instrument. Right-click the *write buffer* input and select *Create» Control* from the shortcut menu.



(c) Place the VISA Read function, located on the *Functions» All Functions» Instrument I/O»VISA* palette, on the block diagram. This function reads data from the instrument. Right-click the *byte count* input and select *Create» Control* from the shortcut menu.

Right-click the *read buffer* output and select *Create» Indicator* from the shortcut menu.



(d) Place the VISA Close function, located on the *Functions» All Functions» Instrument I/O»VISA»VISA Advanced* palette, on the block diagram. This function closes the session with the instrument and releases any system resources that were used.



(e) Place the Simple Error Handler VI, located on the *Functions» All Functions» Time & Dialog* palette, on the block diagram. This VI checks error conditions and opens a dialog box with error information if an error occurs.

**Step 4:** Save the VI as My VISA Write & Read.vi in the C:\Exercises\LabVIEW Basics I directory.

**Step 5:** Display the front panel. Enter devsim in the *VISA resource name* input and set *byte count* to 200 to make sure you read all the information. Type \*IDN? in the *write buffer* and run the VI.

**Step 6:** The top of the instrument simulator lists other commands that are recognized by this instrument. Try other commands in this VI.

**Step 7:** Close the VI when finished.

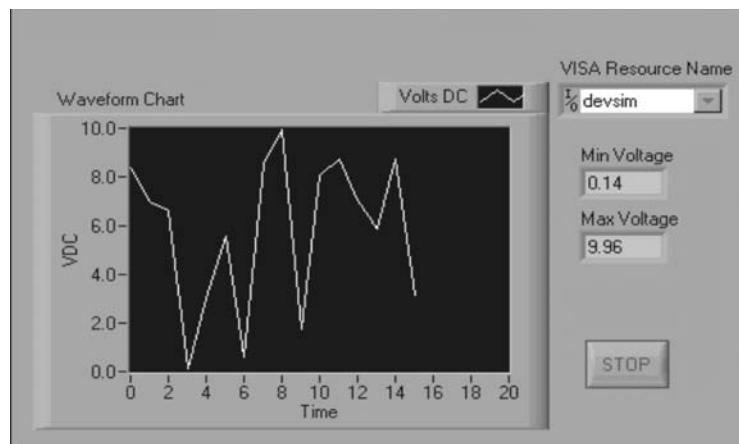
**Problem 10.4** Build a VI that uses the DevSim instrument driver VIs to acquire and plot voltages.

**Solution** Complete the following steps to build a voltage monitor solution VI to problem 10.4 that acquires a DC voltage measurement from the NI Instrument Simulator once every second and plots it in a waveform chart until you click a button. As each value is acquired, the VI compares it with the previous minimum and maximum values. The VI calculates and displays the minimum and maximum values continuously on the front panel.

### Front Panel

**Step 1:** Select *File» New*, then select *Template» Frameworks» Single Loop Application* to open the Single Loop Application template VI.

**Step 2:** Build the front panel shown in Figure P10.4(a).



**Figure P10.4** (a) Front panel.

Use the following guidelines to help you construct the front panel.

- Place a VISA resource name control, located on the *Controls» All Controls »I/O* palette, on the front panel.
- Set the x-axis scale of the waveform chart to show incremental values.

### Block Diagram

**Step 3:** Build the block diagram shown in Figure P10.4(b).

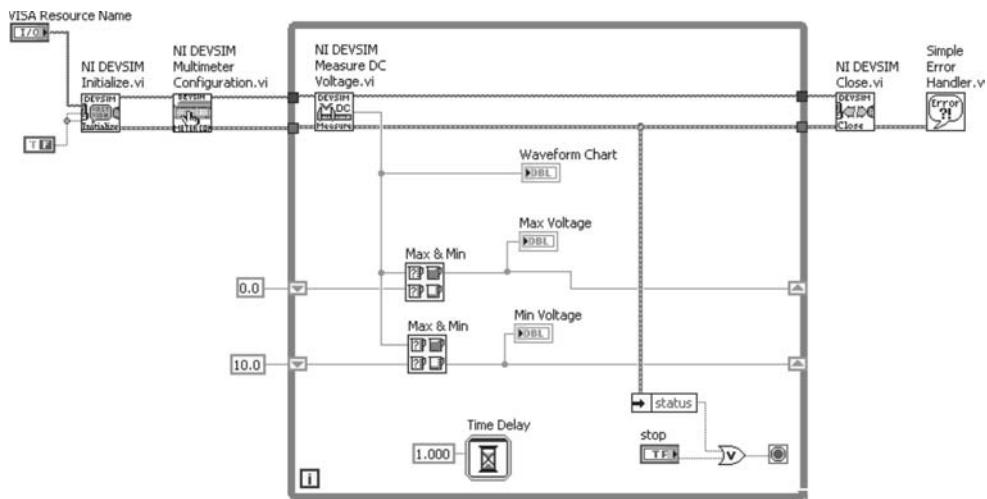


Figure P10.4 (b) Block diagram.

(a) Create two shift registers by right-clicking the right or left border of the loop and selecting *Add Shift Register* from the shortcut menu.



(b) Place the NI DEVSIM Initialize VI, located on the *Functions» Input» Instrument Drivers» NI Device Simulator* palette, on the block diagram. This VI opens communication between LabVIEW and the NI Instrument Simulator.

- Right-click the *ID Query* input and select *Create» Constant* from the shortcut menu. Use the Operating tool to change the constant to a False value.
- Wire the Boolean constant to the *Reset* input.



(c) Place the NI DEVSIM Multimeter Configuration VI, located on the *Functions» Input» Instrument Drivers» NI Device Simulator» Configuration* palette, on the block diagram. This VI configures the range of voltage measurements that the NI Instrument Simulator generates. The default is 0.0 to 10.0 V DC.



(d) Place the NI DEVSIM Measure DC Voltage VI, located on the *Functions» Input» Instrument Drivers» NI Device Simulator» Data* palette, on the block diagram. This VI returns a simulated voltage measurement from the NI Instrument Simulator.



(e) Place the NI DEVSIM Close VI, located on the *Functions» Input» Instrument Drivers» NI Device Simulator* palette, on the block diagram. This VI ends communication between LabVIEW and the NI Instrument Simulator.



(f) Place the Max & Min function, located on the *Functions» All Functions» Comparison* palette, on the block diagram. Use two of these functions to check the current voltage against the minimum and maximum values stored in the shift registers.



(g) Place the Simple Error Handler VI, located on the *Functions »All Functions» Time & Dialog* palette, on the block diagram. This VI displays a dialog box if an error occurs and displays the error information.



(h) Place the *Unbundle by Name* function, located on the *Functions» All Functions» Cluster* palette, on the block diagram. This function accepts *status* from the error cluster.



(i) Place the Or function, located on the *Functions» Arithmetic & Comparison» Express Boolean* palette, on the block diagram. This function controls when the While Loop ends. If there is an error or you click the *Stop* button, the While Loop stops.



(j) Set the wait for the Time Delay Express VI to 1 second.

(k) Wire the block diagram as shown in Figure P10.4(b).

**Note:** You do not need to wire every terminal for each node. Wire only the necessary inputs for each node, such as instrument descriptor, VISA session and error I/O.

**Step 4:** Save the VI as Voltage Monitor.vi in the C:\Exercises\ LabVIEW Basics I directory.

**Step 5:** Make sure the NI Instrument Simulator is powered on.

**Step 6:** Display the front panel and run the VI. The LEDs alternate between Listen and Talk as LabVIEW communicates with the GPIB instrument once a second to get a simulated voltage reading. This voltage displays on the chart, and the minimum and maximum values update accordingly.

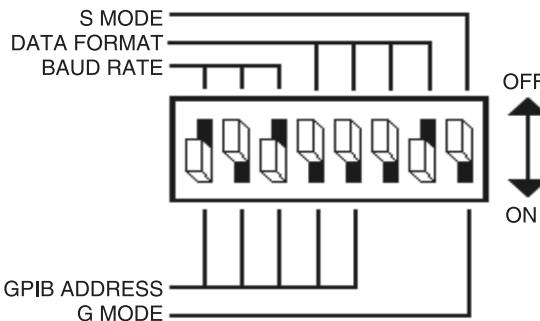
**Step 7:** Stop and close the VI.

**Problem 10.5** Build a Serial Write & Read VI that communicates with an RS-232 device.

**Solution** To get the solution to the problem, complete the following steps to use the Instrument I/O Assistant to build a VI that communicates with the NI Instrument Simulator.

### NI Instrument Simulator

**Step 1:** Power off the NI Instrument Simulator and configure it to communicate through the serial port by setting the switches on the side of the box [Figure P10.5(a)].



**Figure P10.5(a)** Switches of the simulator.

These switch settings configure the instrument as a serial device with the following settings:

- Baud rate = 9,600
- Data bits = 8
- Parity = no parity
- Stop bits = 1
- Flow control parameters = hardware handshaking

Handshaking is a means of data flow control. Software handshaking involves embedding control characters in transmitted data. For example, XON/XOFF flow control works by enclosing a transmitted message between the two control characters XON and XOFF. Hardware handshaking uses voltages on physical wires to control data flow.

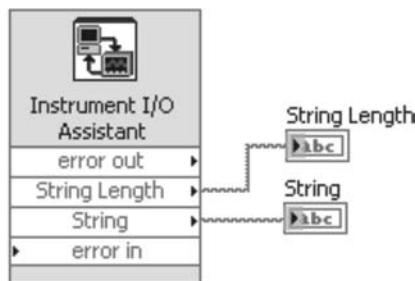
The RTS and CTS lines of the RS-232 device are frequently used for this purpose. Most lab equipment uses hardware handshaking.

**Step 2:** Make sure the NI Instrument Simulator is connected to a serial port on the computer with a serial cable. Make a note of the port number.

**Step 3:** Power on the NI Instrument Simulator. The Power, Ready and Listen LEDs are lit to indicate that the device is in serial communication mode.

### Block Diagram

**Step 4:** Open a blank VI and build the block diagram as shown in Figure P10.5(b).



**Figure P10.5(b)** Block diagram.



(a) Place the Instrument I/O Express VI, located on the *Functions» Input* palette, on the block diagram. Complete the following step in the *Instrument I/O Assistant* dialog box that appears to configure the Express VI.

- (i) Choose *COM1* (or *COM2* depending on the connection port of the NI Instrument Simulator) from the *Select an instrument* pull-down menu.
- (ii) Click the *Add Step* button and click *Write*. In the command field, type *\*IDN?* and select *\n* as the *Termination character*.
- (iii) Click the *Add Step* button and click *Read and Parse*.
- (iv) Click the *Add Step* button and click *Read and Parse* again.

**Note:** The Instrument Simulator returns the byte size of the response, the termination character, the response, then another termination character. Therefore after *\*IDN?* is sent to the instrument, the response must be read twice.

- (v) Click the *Run* button (not the *Run this step* button). The *Run* button runs the entire sequence.
- (vi) Return to the first *Read and Parse* step.
- (vii) Click the *Auto parse* button. The value returned is the size in bytes of the query response.
- (viii) Rename Token to String Length in the *Token name* text box.
- (ix) Select the second *Read and Parse* step.
- (x) Click the *Auto parse* button. The value returned is the identification string of the NI Instrument Simulator.
- (xi) Rename Token to String in the *Token name* text box. The configuration window should be similar to Figure P10.5(c).

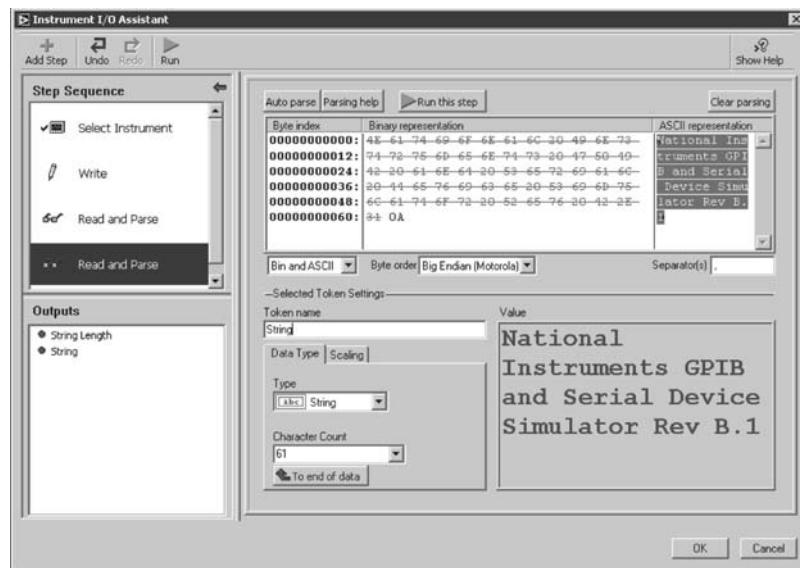


Figure P10.5(c) Configuration window.

- (xii) Select *OK* to return to the block diagram.

(b) Right-click the *String* output and select *Create» Indicator* from the shortcut menu.

(c) Right-click the *String Length* output and select *Create» Indicator* from the shortcut menu.

**Tip** Since LabVIEW is set to handle errors automatically, there is no need to connect a Simple Error Handler VI to *error out*.

**Step 5:** Display the front panel and run the VI.

**Step 6:** Save the VI as *Serial Communication.vi* in the C:\Exercises\LabVIEW Basics I directory.

**Step 7:** Close the VI when finished.

**Problem 10.6** To graph a waveform that an instrument such as a digital oscilloscope returns as an ASCII or binary string.

**Solution** For the ASCII waveform string, the waveform consists of 128 points. Up to four ASCII characters separated by commas represent each point. The following header precedes the data points:

CURVE {12,28, 63,...128 points in total...}CR LF

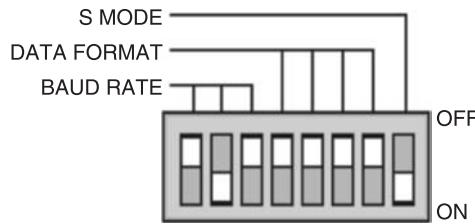
For the binary waveform string, the waveform consists of 128 points. Each point is represented as a 1-byte unsigned integer. The following header precedes the data points:

CURVE % {Bin Count MSB}{Bin Count LSB} {å¤...128 bytes in total...} {Checksum}  
CR LF

Complete the following steps to examine a VI that converts the waveform to an array of numbers. The VI graphs the array and reads the waveform string from the NI Instrument Simulator or from a previously stored array.

### NI Instrument Simulator

**Step 1:** Power off the NI Instrument Simulator and configure it to communicate through the GPIB by setting the switches on the side of the box [Figure P10.6(a)].



**Figure P10.6(a)** Switches of simulator.

These switch settings configure the instrument as a GPIB device with an address of 2.

**Step 2:** Power on the NI Instrument Simulator. Only the Power and Ready LEDs are lit to indicate that the NI Instrument Simulator is in GPIB communication mode.

### Front Panel

**Step 3:** Open the Waveform Example VI located in the C:\Exercises\LabVIEW Basics I directory. The front panel shown in Figure P10.6(b) is already built.

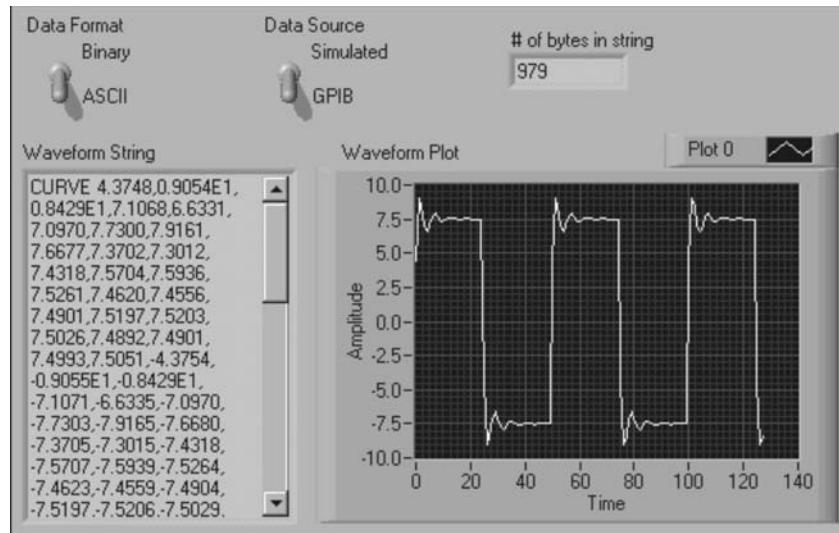


Figure P10.6(b) Front panel.

*Data Format* specifies an ASCII waveform or a binary waveform.

*Data Source* specifies whether the data is simulated or read from the NI Instrument Simulator through the GPIB.

### Block Diagram

**Step 4:** Display and examine the block diagram as shown in Figure P10.6(c).

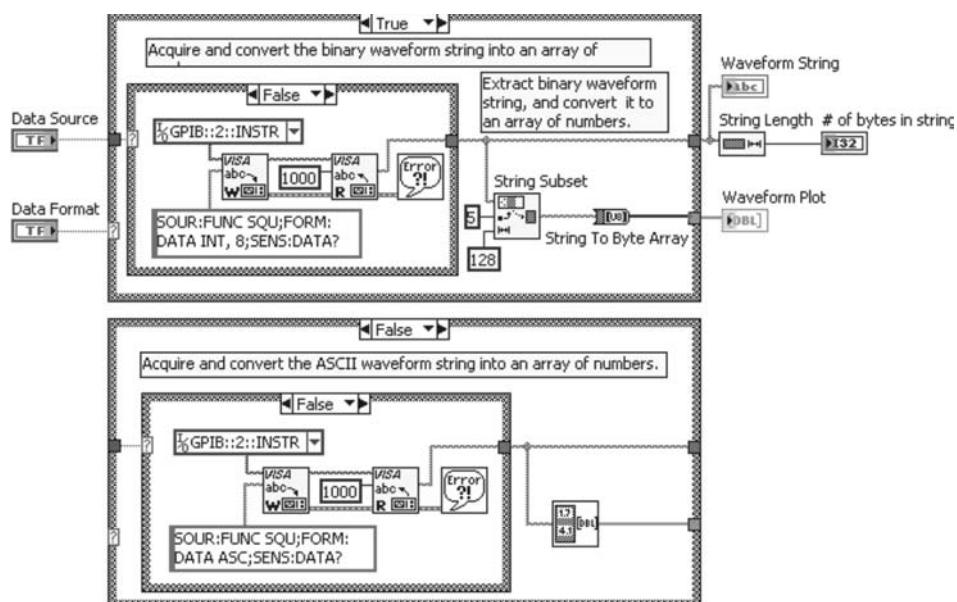


Figure P10.6(c) Block diagram.



(a) The String Subset function located on the *Functions» All Functions» String* palette returns a substring of 128 elements starting from the fifth byte of the binary waveform string, excluding the header and trailer bytes.



(b) The String to Byte Array function, located on the *Functions» All Functions» String» String/Array/Path Conversion* palette, converts the binary string to an array of unsigned integers.



(c) The String Length function, located on the *Functions» All Functions» String* palette, returns the number of characters in the waveform string.



(d) The Extract Numbers VI, located in the Exercises directory, extracts numbers from the ASCII waveform string and places them in an array. Non-numeric characters, such as commas, separate numbers in the string.



(e) The VISA Write and VISA Read functions, located on the *Functions» All Functions» Instrument I/O»VISA* palette, query the NI Instrument Simulator for a square wave in either ASCII or 1-byte binary format.

(f) The Simple Error Handler VI, located on the *Functions» All Functions» Time & Dialog* palette, reports any errors.

**Step 5:** Display the front panel and run the VI. The True case acquires and converts the binary waveform string to an array of numeric values. The False case acquires and converts the ASCII waveform string to an array of numeric values.

**Step 6:** Set *Data Format* to *ASCII* and run the VI. The ASCII waveform string displays, the VI converts the values to a numeric array, and displays the string length and numeric array.

**Step 7:** Set *Data Format* to *Binary* and run the VI again. The binary waveform string and string length display, the VI converts the string to a numeric array, and displays it in the graph.

**Note:** The binary waveform is similar to the ASCII waveform. However, the number of bytes in the string is significantly lower. It is more efficient to transfer waveforms as binary strings than as ASCII strings because binary encoding requires fewer bytes to transfer the same information.

**Step 8:** Close the VI. Do not save changes.

---

## REVIEW QUESTIONS

---

1. What is an Instrument Driver?
2. How do Instrument Drivers work?
3. Which GPIB board does LabVIEW support?
4. What is the difference between GPIB and serial?
5. Explain GPIB communication, configuration and addressing.

6. What is VISA? List its advantages.
7. What is the use of the Instrument I/O Assistant? List the steps to launch it.
8. Enumerate the purpose and advantages of an instrument driver?
9. Draw a typical serial communication and interpret the data bits in a character frame.
10. Explain the common standards of serial port communication.

---

### EXERCISE

---

1. Open the Voltage Monitor VI, which you built in Problem 10.4. Modify the block diagram so that the data are written to a spreadsheet file named voltage.txt in the following format.

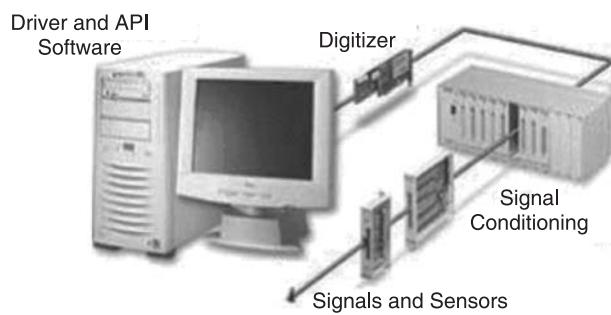
	A	B	C	D
1	Start Date: 6/8/00	Start Time: 1:26 PM		
2	Max Voltage: 9.151000	Min Voltage: 0.354010		
3	Data:			
4	8.965			
5	9.067			
6	0.354			
7	5.08			
8	4.511			
9	3.946			
10	6.446			
:	:			
N	2.293			

Select *File»Save As* to save the VI as Voltage Data to File.vi.

# DATA ACQUISITION

## 11.1 INTRODUCTION

The fundamental task of a DAQ (Data Acquisition) system is to measure or generate real-world physical signals. Data acquisition involves gathering signals from measurement sources and digitizing the signal for storage, analysis and presentation on a personal computer (PC). Data acquisition systems come in many different PC technology forms for great flexibility. Scientists and engineers can choose from PCI, PXI, Compact PCI, PCMCIA, USB, Firewire, parallel, or serial ports for data acquisition in test, measurement, and automation applications. The five components to be considered when building a basic DAQ system as shown in Figure 11.1 are transducers, signals, signal conditioning, DAQ hardware, and driver and application software.



**Figure 11.1** Data acquisition system.

## 11.2 TRANSDUSERS

Data acquisition begins with the physical phenomenon to be measured. This physical phenomenon could be the temperature of a room, the intensity of a light source, the pressure inside a chamber, the force applied to an object, or many other things. An effective DAQ system can measure all of these different phenomena.

A transducer is a device that converts a physical phenomenon into a measurable electrical signal, such as voltage or current. The ability of a DAQ system to measure different phenomena depends on the transducers to convert the physical phenomena into signals measurable by the DAQ hardware. Transducers are synonymous with sensors in DAQ systems. There are specific transducers for many different applications, such as measuring temperature, pressure or fluid flow. Table 11.1 shows a short list of some common transducers and the phenomena they can measure. Different transducers have different requirements for converting phenomena into a measurable signal. Some transducers may require excitation in the form of voltage or current. Other transducers may require additional components and even resistive networks to produce a signal.

**TABLE 11.1** Phenomena and existing transducers

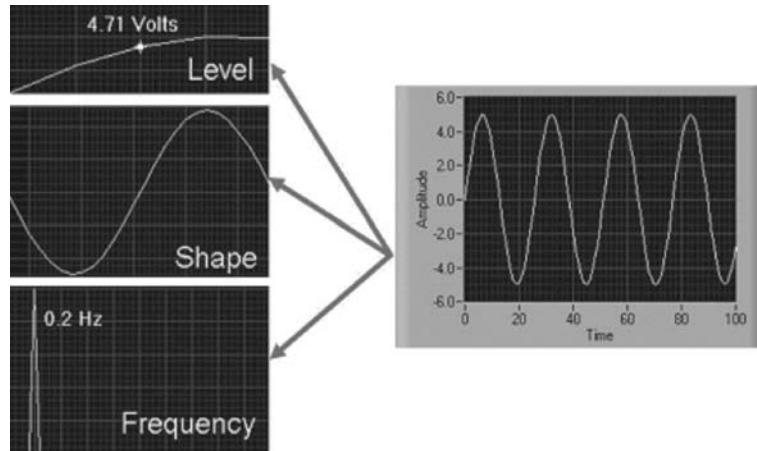
Phenomena	Transducer
Temperature	Thermocouples Resistive Temperature Devices (RTDs) Thermistors
Light	Vacuum Tube Photo Sensors
Sound	Microphone
Force and Pressure	Strain Gauges Piezoelectric Transducers
Position and Displacement	Potentiometers Linear Voltage Differential Transformer Optical Encoder
Fluid	Head Meters Rotational Flowmeters
pH	pH Electrodes

## 11.3 SIGNALS

The appropriate transducer converts the physical phenomena into measurable signals. However, different signals need to be measured in different ways. For this reason, it is important to understand the different types of signals and their corresponding attributes. Signals can be categorized into two groups: *analog* and *digital*.

### 11.3.1 Analog Signals

An analog signal can be at any value with respect to time. A few examples of analog signals include voltage, temperature, pressure, sound and load. The three primary characteristics of an analog signal include level, shape and frequency as shown in Figure 11.2.



**Figure 11.2** Primary characteristics of an analog signal.

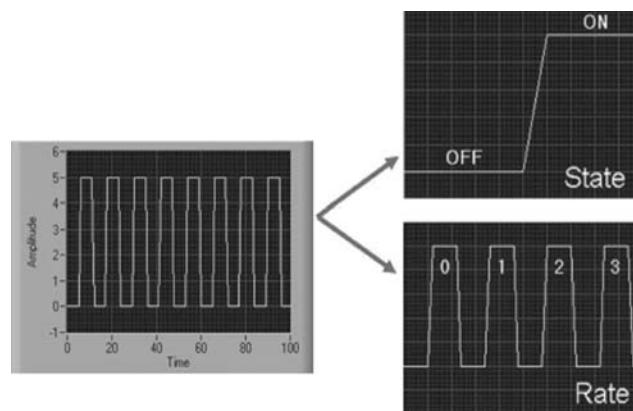
*Level* gives vital information about the measured analog signal since analog signals can take on any value. The intensity of a light source, the temperature in a room, and the pressure inside a chamber are all examples that demonstrate the importance of the level of a signal. When measuring the level of a signal, the signal generally does not change quickly with respect to time. The accuracy of the measurement, however, is very important. A DAQ system that yields maximum accuracy should be chosen to aid in analog level measurements.

Some signals are named after their specific *shape*—sine, square, sawtooth and triangle. The shape of an analog signal can be as important as the level, because measuring the shape of an analog signal allows further analysis of the signal, including peak values, DC values and slope. Signals where the shape is of interest generally change rapidly with respect to time, but system accuracy is still important. The analysis of heartbeats, video signals, sounds, vibrations and circuit responses are some applications involving shape measurements.

All analog signals can be categorized by their *frequency*. Unlike the level or shape of the signal, frequency cannot be directly measured. The signal must be analyzed using software to determine the frequency information. This analysis is usually done using an algorithm known as the *Fourier transform*. When frequency is the most important piece of information, it is important to consider to include both accuracy and acquisition speed. Although the acquisition speed for acquiring the frequency of a signal is less than the speed required for obtaining the shape of a signal, the signal must still be acquired fast enough that the pertinent information is not lost while the analog signal is being acquired. The condition that stipulates this speed is known as the *Nyquist sampling theorem*. Speech analysis, telecommunication and earthquake analysis are some examples of common applications where the frequency of the signal must be known.

### 11.3.2 Digital Signals

A digital signal cannot take on any value with respect to time. Instead, a digital signal has two possible levels: *high* and *low*. Digital signals generally conform to certain specifications that define the characteristics of the signal. Digital signals are commonly referred to as Transistor-to-Transistor Logic (TTL). TTL specifications indicate a digital signal to be low when the level falls within 0 to 0.8 volts, and the signal is high between 2 to 5 volts. The useful information that can be measured from a digital signal includes the state and the rate as shown in Figure 11.3.



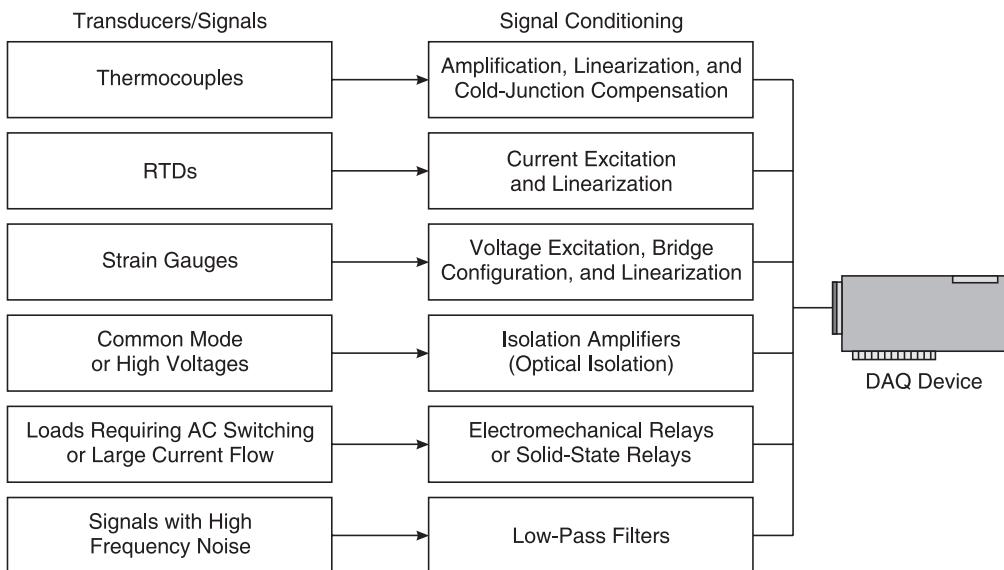
**Figure 11.3** Primary characteristics of a digital signal.

The *state* of a digital signal is essentially the level of the signal—on or off, high or low. Monitoring the state of a switch—open or closed—is a common application showing the importance of knowing the state of a digital signal.

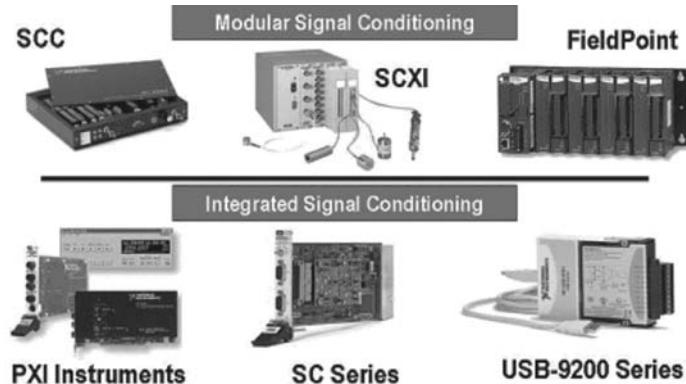
The *rate* of a digital signal defines how the digital signal changes state with respect to time. An example of measuring the rate of a digital signal includes determining how fast a motor shaft spins. Unlike frequency, the rate of a digital signal measures how often a portion of a signal occurs. A software algorithm is not required to determine the rate of a signal.

## 11.4 SIGNAL CONDITIONING

Signal conditioning is the process of measuring and manipulating signals to improve accuracy, isolation, filtering, and so on. Many stand-alone instruments and DAQ devices have built-in signal conditioning. Signal conditioning also can be applied externally, by designing a circuit to condition the signal or by using devices specifically made for signal conditioning. National Instruments has SCXI devices and other devices that are designed for this purpose. Signal conditioning accessories can be used in a variety of important applications. Signal conditioning accessories amplify low-level signals and then isolate and filter them for more accurate measurements. In addition, some transducers use voltage or current excitation to generate a voltage output. Common types of signal conditioning are amplification, isolation, multiplexing, filtering, transducer excitation and linearization. Figure 11.4 shows some common types of transducers and signals and the signal conditioning each requires.



**Figure 11.4** Common transducers and signal conditioning types.



**Figure 11.5** Signal conditioning hardware options.

Sometimes transducers generate signals too difficult or too dangerous to measure directly with a DAQ device. For instance, when dealing with high voltages, noisy environments, extreme high and low signals, or simultaneous signal measurement, signal conditioning is essential for an effective DAQ system. Signal conditioning maximizes the accuracy of a system, allows sensors to operate properly and guarantees safety. It is important to select the right hardware for signal conditioning. Signal conditioning is offered in both modular and integrated forms as shown in Figure 11.5. Signal conditioning accessories can be used in a variety of applications including amplification, attenuation, isolation, bridge completion, simultaneous sampling, sensor excitation, multiplexing, etc. Other important criteria to consider with signal conditioning include packaging (modular versus integrated), performance, I/O count, advanced features, and cost.

### 11.4.1 Amplification

Amplification is the most common type of signal conditioning. Amplifying electrical signals improves accuracy in the resulting digitized signal and reduces the effects of noise. Signals should be amplified as close to the signal source as possible. By amplifying a signal near the device, any noise that attached to the signal is also amplified. Amplifying near the signal source results in the largest signal-to-noise ratio (SNR). For the highest possible accuracy, amplify the signal so the maximum voltage range equals the maximum input range of the analog-to-digital converter (ADC). Low-level thermocouple signals, for example, should be amplified to increase the resolution and reduce noise.

If you amplify the signal at the DAQ device while digitizing and measuring the signal, noise might have entered the lead wire, which decreases SNR. However, if you amplify the signal close to the signal source with an SCXI module, noise has a less destructive effect on the signal, and the digitized representation is a better reflection of the original low-level signal. SCXI has several signal conditioning modules that amplify input signals. The gain is applied to the low-level signals within the SCXI chassis that are located near the transducers, so the module sends only high-level signals to the PC, minimizing the effects of noise on the readings.

### 11.4.2 Isolation

Another common signal conditioning application is isolating the transducer signals from the computer for safety purposes. The system being monitored may contain high-voltage transients that could damage the computer without signal conditioning. An additional reason for isolation is ensuring that the readings from the plug-in DAQ device are unaffected by differences in ground potentials or common-mode voltages. When the DAQ device input and the signal being acquired are each referenced to ‘ground’, problems occur if there is a potential difference in the two grounds. This difference can lead to what is known as a ground loop, which may cause inaccurate representation of the acquired signal; or if the difference is too large, it may damage the measurement system. Using isolated signal conditioning modules eliminates ground loops and ensures that the signals are accurately acquired.

You also can use isolation to ensure that differences in ground potentials do not affect measurements from the DAQ device. When you do not reference the DAQ device and the signal to the same ground potential, a ground loop can occur. Ground loops can cause an inaccurate representation of the measured signal. If the potential difference between the signal ground and the DAQ device ground is large, damage can occur to the measuring system. Isolating the signal eliminates the ground loop and ensures that the signals are accurately measured.

### 11.4.3 Multiplexing

A common technique for measuring several signals with a single measuring device is multiplexing. Signal conditioning hardware for analog signals often provides multiplexing for use with slowly changing signals like temperature. The ADC samples one channel, switches to the next channel, samples it, switches to the next channel, and so on. Because the same ADC samples many channels instead of one, the effective sampling rate of each individual channel is inversely proportional to the number of channels sampled. For example, a PCI-MIO-16E-1 sampling at 1 MS/s on 10 channels will effectively sample each individual channel at:

$$\frac{1 \text{ MS/s}}{10 \text{ channels}} = 100 \text{ kS/s per channel}$$

SCXI modules for analog signals employ multiplexing so that as many as 3,072 signals can be measured with one DAQ device. With the AMUX-64T analog multiplexer, you can measure up to 256 signals with a single device. This feature is in addition to any built-in multiplexing on the DAQ device.

#### **11.4.4 Filtering**

The purpose of a filter is to remove unwanted signals from the signal that you are trying to measure. A noise filter is used on DC-class signals, such as temperature, to attenuate higher frequency signals that can reduce your measurement accuracy. For example, many SCXI modules use 4 Hz and 10 kHz lowpass filters to eliminate noise before the signals are digitized by the DAQ device.

AC-class signals, such as vibration, often require a different type of filter known as an antialiasing filter. Like the noise filter, the antialiasing filter is also a lowpass filter; however, it requires a very steep cutoff rate, so it almost completely removes all signal frequencies that are higher than the input bandwidth of the device. If the signals were not removed, they would erroneously appear as signals within the input bandwidth of the device. Devices designed specifically for AC-class signal measurement – the NI 455x, NI 445x, and NI 447x dynamic signal acquisition (DSA) devices, the NI 61xx simultaneous-sampling multifunction I/O devices, and the SCXI-1141 module have built-in antialiasing filters.

#### **11.4.5 Transduces Excitation**

Signal conditioning systems can generate excitation, which some transducers require for operation. Strain gauges and RTDs require external voltage and currents, respectively, to excite their circuitry into measuring physical phenomena. This type of excitation is similar to a radio that needs power to receive and decode audio signals. Signal conditioning modules for these transducers usually provide these signals. RTD measurements are usually made with a current source that converts the variation in resistance to a measurable voltage. Strain gauges, which are very low-resistance devices, typically are used in a Wheatstone bridge configuration with a voltage excitation source. The SCXI-1121 and SCXI-1122 have onboard excitation sources, configurable as current or voltage that you can use for strain gauges, thermistors, or RTDs.

#### **11.4.6 Linearization**

Another common signal conditioning function is linearization. Many transducers, such as thermocouples, have a nonlinear response to changes in the physical phenomena you measure. LabVIEW can linearize the voltage levels from transducers so you can scale the voltages to the measured phenomena. LabVIEW provides scaling functions to convert voltages from strain gages, RTDs, thermocouples, and thermistors. You should understand the nature of your signal, the configuration that is being used to measure the signal, and the effects of the environment surrounding the system. Based on this information, you can determine whether signal conditioning will be a necessary part of your DAQ system.

## 11.5 DAQ HARDWARE CONFIGURATION

Before using a data acquisition board, you must confirm that the software can communicate with the board by configuring the devices. The Windows Configuration Manager keeps track of all the hardware installed in the computer, including National Instruments DAQ devices. If you have a Plug & Play (PnP) device, such as an E Series MIO device, the Windows Configuration Manager automatically detects and configures the device. If you have a non-PnP device, or legacy device, you must configure the device manually using the Add New Hardware option in the Control Panel. You can verify the Windows Configuration by accessing the Device Manager.

### 11.5.1 Measurement & Automation Explorer

LabVIEW installs Measurement & Automation Explorer (MAX) which establishes all device and channel configuration parameters. After installing a DAQ device in the computer, you must run this configuration utility. MAX reads the information the Device Manager records in the Windows Registry and assigns a logical device number to each DAQ device. Use the device number to refer to the device in LabVIEW. Access MAX either by double-clicking the icon on the desktop or selecting **Tools»Measurement & Automation Explorer** in LabVIEW. The device parameters that you can set using the configuration utility depend on the device. MAX saves the logical device number and the configuration parameters in the Windows Registry. The plug and play capability of Windows automatically detects and configures switchless DAQ devices, such as the PCI-6024E. When you install a device in the computer, the device is automatically detected.

Measurement & Automation Explorer, or MAX, is a software interface that gives you access to all National Instruments DAQ, GPIB, IMAQ, IVI, Motion, VISA, and VXI devices connected to your system. The shortcut to MAX is placed on the desktop during installation of NI-DAQ. MAX is used primarily to configure and test National Instruments hardware, but it offers other functionality, such as checking to see if you have the latest version of NI-DAQ installed. The functionality of MAX is divided into four categories—Data Neighborhood, Devices and Interfaces, Scales and Software.

### 11.5.2 Scales

You can configure custom scales for your measurements. This is very useful when working with sensors. It allows you to bring a scaled value into your application without having to work directly with the raw values. For example, in the exercises you use a temperature sensor that represents temperature with a voltage. The conversion equation for the temperature is:  $\text{Voltage} \times 100 = \text{Celsius}$ . After a scale is set, you can use it in your application program, providing the temperature value, rather than the voltage.

### 11.5.3 Simulating a DAQ Device

You can create NI-DAQmx simulated devices in NI-DAQmx 7.4 or later. Using NI-DAQmx simulated devices, you can try NI products in your application without the hardware. When you later acquire the hardware, you can import the NI-DAQmx simulated device configuration to the physical device using the MAX Portable Configuration Wizard. With NI-DAQmx simulated devices, you also can export a physical device configuration onto a system that does not have the physical

device installed. Then, using the NI-DAQmx simulated device, you can work on your applications on a portable system and upon returning to the original system, you can easily import your application work.

To create an NI-DAQmx simulated device, complete the following steps:

**Step 1:** Right-click *Devices and Interfaces* and select *Create»New*.

**Step 2:** A dialog box prompts you to select a device to add. Select *NI-DAQmx Simulated Device* and click *Finish*.

**Step 3:** In the *Choose Device* dialog box, select the family of devices for the device you want to simulate.

**Step 4:** Select the device and click *OK*. In the configuration tree in MAX, the icons for NI-DAQmx simulated devices are yellow. The icons for physical devices are green.

**Step 5:** If you select a PXI device, you are prompted to select a chassis number and PXI slot number.

**Step 6:** If you select an SCXI chassis, the SCXI configuration panels open.

To remove an NI-DAQmx simulated device, complete the following steps:

**Step 1:** Expand *Devices and Interfaces»NI-DAQmx Devices*.

**Step 2:** Right-click the NI-DAQmx simulated device you want to delete.

**Step 3:** Click *Delete*.

## 11.6 DAQ HARDWARE

The DAQ hardware acts as the interface between the computer and the outside world. It primarily functions as a device that digitizes incoming analog signals so that the computer can interpret them. Other data acquisition functionality includes Analog Input/Output, Digital Input/Output, Counter/Timers and Multifunction which is a combination of analog, digital, and counter operations on a single device. National Instruments offers several hardware platforms for data acquisition. The most readily available platform is the desktop computer. National Instruments offers PCI DAQ devices that plug into any desktop computer. In addition, NI makes DAQ devices for PXI/CompactPCI, a more rugged modular computer platform specifically for measurement and automation applications. For distributed measurements, National Instruments Compact FieldPoint platform delivers modular I/O, embedded operation and Ethernet communication. For portable or handheld measurements, National Instruments DAQ devices for USB and PCMCIA work with laptops or Pocket PC PDAs as shown in Figure 11.6.

A typical desktop DAQ system has three basic types of hardware—a terminal block, a cable and a DAQ device as shown in Figure 11.7.

After you have converted a physical phenomenon into a measurable signal with or without signal conditioning, you need to acquire that signal. To acquire a signal, you need a terminal block, a cable, a DAQ device and a computer. This hardware combination can transform a standard computer into a measurement and automation system.

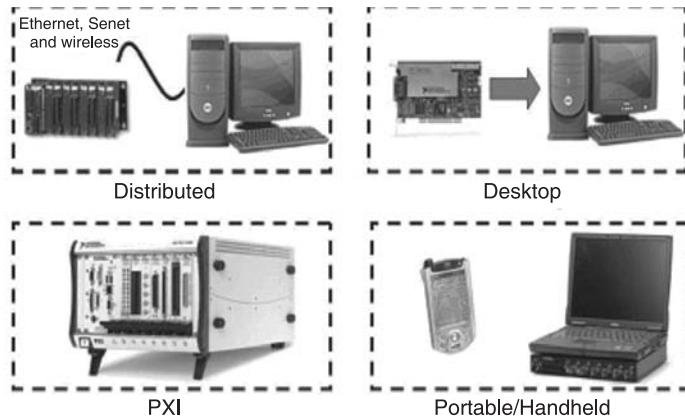


Figure 11.6 DAQ hardware options.

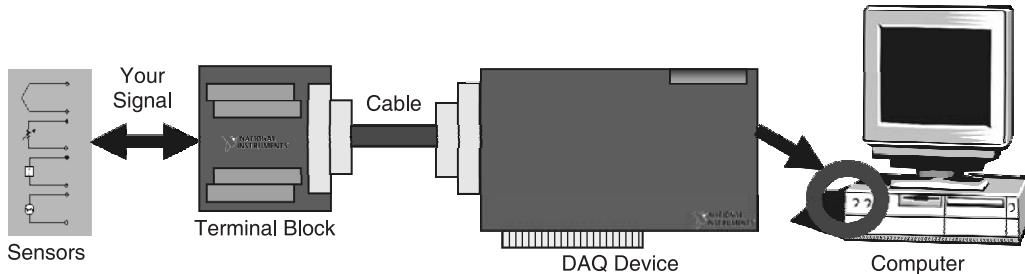


Figure 11.7 Typical DAQ system.

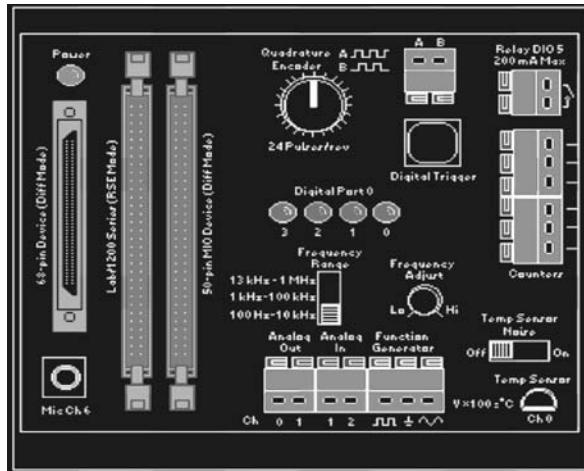
### 11.6.1 Terminal Block and Cable

A terminal block provides a place to connect signals. It consists of screw or spring terminals for connecting signals and a connector for attaching a cable to connect the terminal block to a DAQ device. Terminal blocks have 100, 68, or 50 terminals. The type of terminal block you should choose depends on two factors—the device and the number of signals you are measuring. A terminal block with 68 terminals offers more ground terminals to connect a signal to than a terminal block with 50 terminals. Having more ground terminals prevents the need to overlap wires to reach a ground terminal, which can cause interference between the signals.

### 11.6.2 DAQ Signal Accessory

Figure 11.8 shows the DAQ Signal Accessory. The DAQ Signal Accessory is a customized terminal block designed for learning purposes. It has 3 Connectors, Quadrature Encoder, Relay, Digital Trigger, 4 LEDs (reverse logic), Counter I/O, Function Generator, Function Generator Frequency Control, Temperature Sensor, Temperature Sensor Noise Control, Analog Input and Analog Output.

The three different cable connectors accommodate many different DAQ devices and spring terminals to connect signals. You can access three analog input channels, one of which is connected to the temperature sensor and two analog output channels.



**Figure 11.8** DAQ signal accessory.

The DAQ Signal Accessory includes a function generator with a switch to select the frequency range of the signal, and a frequency knob. The function generator can produce a sine wave or a square wave. A connection to ground is located between the sine wave and square wave terminal.

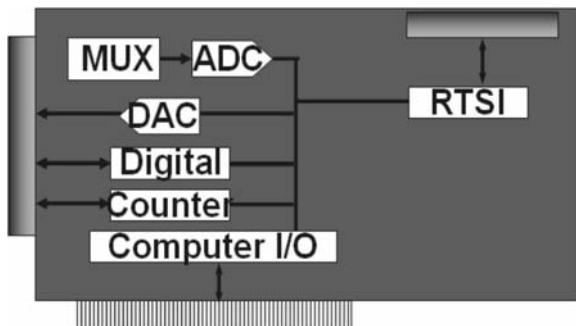
A digital trigger button produces a TTL pulse for triggering analog input or output. When you press the trigger button, the signal goes from +5 V to 0 V when pressed and returns to +5 V when you release the button. Four LEDs connect to the first four digital lines on the DAQ device. The LEDs use reverse logic, so when the digital line is high, the LED is off and vice versa. The DAQ Signal Accessory has a quadrature encoder that produces two pulse trains when you turn the encoder knob. Terminals are provided for the input and output signals of two counters on the DAQ device. The DAQ Signal Accessory also has a relay, a thermocouple input and a microphone jack. The exercises on DAQ will be based on the terminal block.

### 11.6.3 DAQ Device

Before a computer-based measurement system can measure a physical signal such as temperature, a sensor or transducer, must convert the physical or real world signal into an electrical one such as voltage or current. You must use signal conditioning accessories to condition the signals before the plug-in DAQ device converts them to digital information. The software controls the DAQ system by acquiring the raw data, analyzing and presenting the results.

Consider the following options for a DAQ system:

- The plug-in DAQ device shown in Figure 11.9 resides in the computer. You can plug the device into the PCI slot of a desktop computer or the PCMCIA slot of a laptop computer for a portable DAQ measurement system.
- The DAQ device is external and connects to the computer through an existing port, such as the serial port or Ethernet port, which means you can quickly and easily place measurement nodes near sensors.



**Figure 11.9** DAQ device.

The computer receives raw data through the DAQ device. The application you write presents and manipulates the raw data in a form you can understand. The software also controls the DAQ system by commanding the DAQ device when and from which channels to acquire data. Typically, DAQ software includes drivers and application software. Drivers are unique to the device or type of device and include the set of commands the device accepts. Application software, such as LabVIEW, sends the drivers commands, such as acquire and return a thermocouple reading. The application software also displays and analyzes the acquired data. NI measurement devices include NI-DAQ driver software, a collection of VIs you use to configure, acquire data from and send data to the measurement devices.

Most DAQ devices have four standard elements: analog input, analog output, digital I/O, and counters. You can transfer the signal you measure with the DAQ device to the computer through a variety of different bus structures. For example, you can use a DAQ device that plugs into the PCI bus of a computer, a DAQ device connected to the PCMCIA socket of a laptop, or a DAQ device connected to the USB port of a computer. You also can use PXI/CompactPCI to create a portable, versatile, and rugged measurement system. If you do not have a DAQ device, you can simulate one in Measurement and Automation Explorer to complete your software testing.

I/O Connector connects your signal (via terminal block and cable) to the DAQ device. Computer I/O Interface Circuitry connects the DAQ device to the computer. It can be a variety of bus structures PCI, PXI/Compact PCI, ISA/AT, PCMCIA, USB, IEEE 1394 (Firewire). Real-Time System Integration (RTSI) Bus is used to synchronize multiple DAQ devices and allows sharing of timing and trigger signals between devices.

Analog Input Circuitry has a multiplexer (mux). This switch has multiple input channels but only lets one at a time through to the instrumentation amplifier. The instrumentation amplifier either amplifies or attenuates your signal.

Analog-to-Digital Converter (ADC) converts an analog signal to a digital number and is used for analog input. The applications are circuit testing, power supply testing, dynamometer testing, weather station, geophysical studies and filter analysis.

Digital-to-Analog Converter (DAC) converts a digital number to an analog signal and is used for analog output. The applications are control systems, function generator, tone generator and servo motor control.

Digital I/O Circuitry can input or output digital signals and is not suitable for measuring rate. Applications include switch sensing, relay control and controlling LEDs.

Counter Circuitry can input or output digital signals. It is suitable for measuring rate and has built in timing signals. Applications includes stepper motor control, measuring frequency of a rotating shaft and oscillator testing.

Depending on your application, there are several different classes of PC-based data acquisition devices that you can use:

- Analog Input/Output
- Digital Input/Output
- Counter/Timers
- Multifunction—a combination of analog, digital and counter operations

## 11.7 ANALOG INPUTS

Analog input is the process of measuring an analog signal and transferring the measurement to a computer for analysis, display or storage. An analog signal is a signal that varies continuously. Analog input is most commonly used to measure voltage or current. You can use many types of devices to perform analog input, such as multifunction DAQ (MIO) devices, high-speed digitizers, digital multimeters (DMMs) and Dynamic Signal Acquisition (DSA) devices.

### 11.7.1 Analog-to-Digital Conversion

Acquiring an analog signal with a computer requires a process known as *analog-to-digital conversion* which takes an electrical signal and translates it into digital data so that a computer can process it. *Analog-to-digital converters* (ADCs) are circuit components that convert a voltage level into a series of ones and zeroes. ADCs sample the analog signal on each rising or falling edge of a sample clock. In each cycle, the ADC takes a snapshot of the analog signal, so that the signal can be measured and converted into a digital value. A *sample clock* controls the rate at which samples of the input signal are taken. Because the incoming, or unknown signal is a real world signal with infinite precision, the ADC approximates the signal with fixed precision. After the ADC obtains this approximation, the approximation can be converted to a series of digital values. Some conversion methods do not require this step, because the conversion generates a digital value directly as the ADC reaches the approximation.

### 11.7.2 Task Timing

When performing analog input, the task can be timed to Acquire 1 Sample, Acquire *n* Samples or Acquire Continuously.

Acquiring a *single sample* is an on-demand operation. In other words, the driver acquires one value from an input channel and immediately returns the value. This operation does not require any buffering or hardware timing. For example, if you periodically monitor the fluid level in a tank, you would acquire single data points. You can connect the transducer that produces a voltage representing the fluid level to a single channel on the measurement device and initiate a single-channel, single-point acquisition when you want to know the fluid level.

One way to acquire *multiple samples* for one or more channels is to acquire single samples in a repetitive manner. However, acquiring a single data sample on one or more channels over and

over is inefficient and time consuming. Moreover, you do not have accurate control over the time between each sample or channel. Instead you can use hardware timing, which uses a buffer in computer memory, to acquire data more efficiently. Programmatically, you need to include the timing function and specify the *sample rate* and the *sample mode (finite)*. As with other functions, you can acquire multiple samples for a single channel or multiple channels. With NI-DAQmx, you also can gather data from multiple channels. For instance, you might want to monitor both the fluid level in the tank and the temperature. In such a case, you need two transducers connected to two channels on the device.

If you want to view, process or log a subset of the samples as they are acquired, you need to continually acquire samples. For these types of applications, set the sample mode to *continuous*.

### 11.7.3 Task Triggering

When a device controlled by NI-DAQmx does something, it performs an action. Two very common actions are producing a sample and starting a waveform acquisition. Every NI-DAQmx action needs a stimulus or cause. When the stimulus occurs, the action is performed. Causes for actions are called *triggers*. The start trigger starts the acquisition. The reference trigger establishes the reference point in a set of input samples. Data acquired up to the reference point is pretrigger data. Data acquired after the reference point is posttrigger data.

## 11.8 ANALOG OUTPUTS

Analog output is the process of generating electrical signals from your computer. It is generated by performing digital-to-analog (D/A) conversions. The available analog output types for a task are voltage and current. To perform a voltage or current task, a compatible device must be installed that can generate that form of signal.

### 11.8.1 Task Timing

When performing analog output, the task can be timed to Generate 1 Sample, Generate *n* Samples or Generate Continuously.

To Generate 1 Sample use single updates if the signal level is more important than the generation rate. For example, generate one sample at a time if you need to generate a constant, or DC signal. You can use software timing to control when the device generates a signal. This operation does not require any buffering or hardware timing. For example, if you need to generate a known voltage to stimulate a device, a single update would be an appropriate task.

One way to generate *n* samples or multiple samples for one or more channels is to generate single samples in a repetitive manner. However, generating a single data sample on one or more channels over and over is inefficient and time consuming. Moreover, you do not have accurate control over the time between each sample or channel. Instead, you can use hardware timing which uses a buffer in computer memory to generate samples more efficiently. You can use software timing or hardware timing to control when a signal is generated. With software timing, the rate at which the samples are generated is determined by the software and operating system instead of by the measurement device. With hardware timing, a TTL signal, such as a clock on the device,

controls the rate of generation. A hardware clock can run much faster than a software loop. A hardware clock is also more accurate than a software loop. Programmatically, you need to include the timing function, specifying the *sample rate* and the *sample mode (finite)*. As with other functions, you can generate multiple samples for a single channel or multiple channels. Use Generate *n* Samples if you want to generate a finite time-varying signal, such as an AC sine wave.

Continuous generation is similar to Generate *n* Samples, except that an event must occur to stop the generation. If you want to continuously generate signals, such as generating a non-finite AC sine wave, set the timing mode to *continuous*.

### 11.8.2 Task Triggering

When a device controlled by NI-DAQmx does something, it performs an action. Two very common actions are producing a sample and starting a generation. Every NI-DAQmx action needs a stimulus or cause. When the stimulus occurs, the action is performed. Causes for actions are called triggers. A start trigger starts the generation.

### 11.8.3 Digital-to-Analog Conversion

Digital-to-analog conversion is the opposite of analog-to-digital conversion. In digital-to-analog conversion, the data starts in the computer. The data might have been acquired earlier using analog input or may have been generated by software on the computer. A digital-to-analog converter (DAC) accepts this data and uses it to vary the voltage on an output pin over time. The DAC generates an analog signal that the DAC can send to other devices or circuits. A DAC has an update clock that tells the DAC when to generate a new value. The function of the update clock is similar to the function of the sample clock for an analog-to-digital converter (ADC). At each cycle the clock, the DAC converts a digital value to an analog voltage and creates an output as a voltage on a pin. When used with a high speed clock, the DAC can create a signal that appears to vary constantly and smoothly.

## 11.9 COUNTERS

A counter is a digital timing device. Counters are used for event counting, frequency measurement, period measurement, position measurement and pulse generation. Pulse generation counter/timer circuitry is useful for many applications, including counting the occurrences of a digital event, digital pulse timing, and generating square waves and pulses. You can implement all these applications using three counter/timer signals—gate, source and output.

**Count register**—It stores the current count of the counter. You can query the count register with software.

**Source**—It is an input signal that can change the current count stored in the count register. The counter looks for rising or falling edges on the source signal. Whether a rising or falling edge changes, the count is software selectable. The type of edge selected is referred to as the active edge of the signal. When an active edge is received on the source signal, the count changes. Whether an active edge increments or decrements the current count is also software selectable.

**Gate**—It is an input signal that determines if an active edge on the source changes the count. Counting can occur when the gate is high, low, or between various combinations of rising and falling edges. Gate settings are made in software.

**Output**—It is an output signal that generates pulses or a series of pulses, otherwise known as a *pulse train*.

When you configure a counter for simple event counting, the counter increments when an active edge is received on the source. In order for the counter to increment on an active edge, the counter must be armed or started. A counter has a fixed number it can count to as determined by the resolution of the counter. The most significant specifications for operation of a counter/timer are the resolution and clock frequency. The resolution is the number of bits the counter uses. A higher resolution simply means that the counter can count higher. The clock frequency determines how fast you can toggle the digital source input. With higher frequency, the counter increments faster and therefore can detect higher frequency signals on the input and generate higher frequency pulses and square waves on the output. The DAQ-STC counter/timer used on our E Series DAQ devices, for example, has 16 and 24-bit counters with a clock frequency of 20 MHz. The NI-TIO counter/timer used on NI 660x counter/timer devices has eight 32-bit counters with a maximum clock frequency of 80 MHz.

## 11.10 DIGITAL I/O (DIO)

DIO interfaces are often used on PC DAQ systems to control processes, generate patterns for testing and communicate with peripheral equipment. In each case, the important parameters include the number of digital lines available, the rate at which you can accept and source digital data on these lines and the drive capability of the lines. If the digital lines are used for controlling events such as turning on and off heaters, motors or lights, a high data rate is usually not required because the equipment cannot respond very quickly. The number of digital lines, of course, should match the number of processes to be controlled. In each of these examples, the amount of current required to turn the devices on and off must be less than the available drive current from the device. DIO can also be used in industrial applications to verify that a switch is open or closed and to check the voltage levels as high or low. It can also be used for high-speed handshaking or simple communication methods.

With the proper digital signal conditioning accessories, you can use the low-current TTL signals to/from the DAQ hardware to monitor/control high voltage and current signals from industrial hardware or to drive external relays. For example, the voltage and current needed to open and close a large valve may be on the order of 100 VAC at 2 A. Because the output of a DIO device is 0 to 5 VDC at several milliamperes, a signal conditioning module such as SCXI is needed to switch the power signal to control the valve.

A common DIO application is to transfer data between a computer and equipment such as data loggers, data processors and printers. Because this equipment usually transfers data in one byte (8-bit) increments, the digital lines on a plug-in DIO device are arranged in groups of eight. In addition, some devices with digital capabilities will have handshaking circuitry for communication-synchronization purposes. The number of channels, data rate, and handshaking

capabilities are all important specifications that should be understood and matched to the application needs.

## 11.11 DAQ SOFTWARE ARCHITECTURE

National Instruments data acquisition boards have a driver engine that communicates between the board and the application software. There are two different driver engines the NI-DAQmx and Traditional NI-DAQ. You can use LabVIEW to communicate with these driver engines. The DAQ Assistant in LabVIEW can be used to communicate with your data acquisition board. The DAQ Assistant is an Express VI that communicates with NI-DAQmx. Measurement & Automation Explorer (MAX) is useful for configuring and testing your data acquisition boards. Driver software is the layer of software that directly programs the registers of the DAQ hardware, managing its operation and its integration with the computer resources like processor interrupts and memory. Driver software hides the low-level, complicated details of hardware programming, providing the user with an easy-to-understand interface or a stand-alone application program.

### 11.11.1 Driver Software

Software transforms the PC and the DAQ hardware into a complete data acquisition, analysis and presentation tool. Without software to control or drive the hardware, the DAQ device will not work properly. Driver software is the layer of software that allows easy communication to the hardware. It forms the middle layer between the application software and the hardware. Driver software also prevents a programmer from having to do register-level programming or complicated commands in order to access the hardware functions. National Instruments offers two different software options, one is the NI-DAQmx driver and additional measurement services software and the second is the NI-DAQmx base driver software.

With the introduction of NI-DAQmx, National Instruments revolutionized DAQ application development by greatly reducing the speed at which you can move from building a program to deploying a high-performance measurement application. The DAQ Assistant, which is included with NI-DAQmx, is a graphical, interactive guide for configuring, testing and acquiring measurement data. With a single click, you can even generate code based on your configuration, making it easier and faster to develop complex operations. Because the DAQ Assistant is completely menu-driven, you will make fewer programming errors and drastically decrease the time from setting up your DAQ system to taking your first measurement. The increasing sophistication of DAQ hardware, computers and software continues to emphasize the importance and value of good driver software. Driver has the ability to:

- Test channels without any programming
- Acquire data in the background while processing in the foreground
- Use programmed I/O, interrupts and DMA to transfer data
- Stream data to and from disk
- Perform several functions simultaneously
- Integrate multiple DAQ devices
- Integrate seamlessly with sensors and a variety of signal types
- Provide examples to help get started

### 11.11.2 Application Software

The application layer can be either a development environment in which you build a custom application that meets specific criteria, or it can be a configuration-based program with preset functionality. Application software adds analysis and presentation capabilities to the driver software. To choose the right application software, evaluate the complexity of the application, the availability of configuration-based software that fits the application, and the amount of time available to develop the application. If the application is complex or there is no existing program, use a development environment.

NI offers three development environment software products for developing complete instrumentation, acquisition and control applications. One is the LabVIEW with graphical programming methodology, the second is the LabWindows/CVI for traditional C programmers, and the third is the Measurement Studio for Visual Basic, C++ and .NET.

With the introduction of SignalExpress, NI has introduced a configuration-based software environment where programming is no longer a requirement. SignalExpress allows for interactive measurements with NI Express Technology. Additionally, all products can be augmented with add-on toolkits for special functionality. National Instruments VI Logger is an easy-to-use yet very flexible tool specifically designed for your data logging applications.

## 11.12 DAQ ASSISTANT

The DAQ Assistant is a graphical interface for interactively creating, editing, and running NI-DAQmx virtual channels and tasks. A NI-DAQmx virtual channel consists of a physical channel on a DAQ device and the configuration information for this physical channel such as input range and custom scaling. A NI-DAQmx task is a collection of virtual channels, timing and triggering information, and other properties regarding the acquisition or generation. DAQ Assistant provides an interactive guide to configuring, testing and acquiring measurement data. With a single click, you can even generate code based on your configuration, making it easier and faster to develop complex operations. DAQ Assistant is completely menu-driven and you will encounter fewer errors. It drastically decreases the time to your first measurement.

When you place this Express VI on the block diagram, the DAQ Assistant launches to create a new task. After you create a task, you can double-click the DAQ Assistant Express VI to edit that task. For continuous measurement or generation, place a While Loop around the DAQ Assistant Express VI. Whenever a DAQmx Express VI is to be used in a While Loop, it is good practice to wire the stop condition into the *stop* input of the DAQmx Express VI.

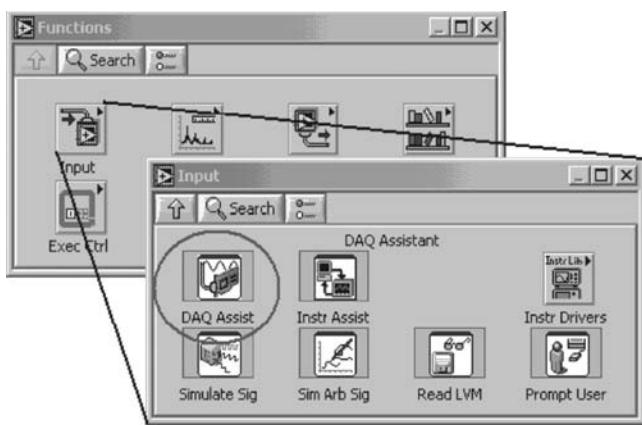
Using the DAQ Assistant Express VI creates a task accessible only to the Express VI. To make the task globally accessible from any application, you must convert the Express VI to an NI-DAQmx task saved in MAX. In LabVIEW 8.0 and later, you can generate NI-DAQmx API code from DAQ Assistant Express VI. Right-click the DAQ Assistant Express VI and select *Generate NI-DAQmx Code* from the shortcut menu to generate both configuration and example code for the task. For continuous single-point input or output, the DAQ Assistant Express VI might not provide optimal performance. In NI-DAQmx, virtual channels are integral to every measurement.

### 11.12.1 Launch the DAQ Assistant

You can launch the DAQ Assistant in several ways. Complete the following steps to launch the DAQ Assistant from a LabVIEW block diagram.

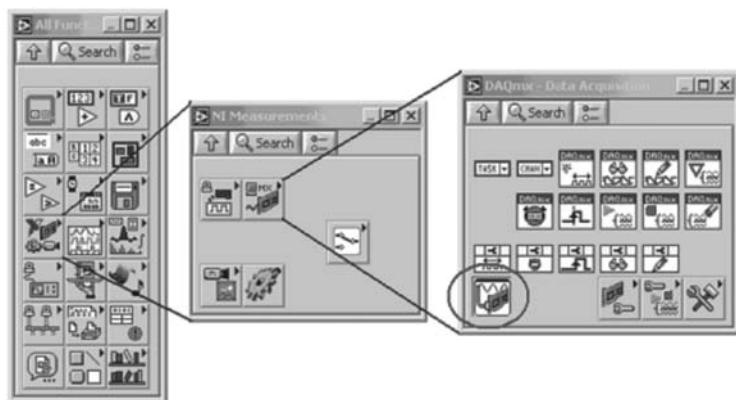
**Step 1:** Open LabVIEW and create a New VI. Switch to the block diagram (Ctrl+E).

**Step 2:** DAQ Assistant Express VI is located in the *Input* subpalette of the *Functions* palette as in Figure 11.10. Place the DAQ Assistant on the block diagram by dragging and dropping it from the *Functions* palette. The Assistant should automatically launch when you drop the VI on the diagram.



**Figure 11.10** Launch the DAQ Assistant located in the Input subpalette.

**Step 3:** It is also available at *Express>>Output>>DAQ Assistant*. In the *Advanced Functions* palette, the DAQ Assistant Express VI is located in the *NI Measurements >> DAQmx* sub-palette as in Figure 11.11. The *Create New* window opens up for task configuration when the DAQ Assistant is placed on the block diagram. Measurement type can be Analog Input, Analog Output, Counter Input, Counter Output and Digital I/O.



**Figure 11.11** Launch the DAQ Assistant located in DAQmx sub-palette.

Once you have located the DAQ Assistant Express VI in the appropriate location, select it from the palette and drop it on the block diagram of your VI. By default, the properties page should pop up, allowing you to configure your task. The first step is to select your type of measurement.

### 11.12.2 Create the Task

We will configure a simple voltage analog input measurement as in Figure 11.12. The analog input task is specific to the measurement. For other measurement and signal generation types, you would follow similar steps.

**Step 1:** On the first screen, select *Analog Input* for your Measurement Type.

**Step 2:** Next, select *Voltage*.

**Step 3:** The next screen lets you select the physical channel (or channels) for which you are creating this task. All supported data acquisition hardware devices should appear in the tree control and you can expand them to view a list of the physical channels that you can select for your task. To select more than one channel, hold down the *Ctrl* button while clicking on the channel names.

**Step 4:** Click *Finish* to move on to the configuration stage.

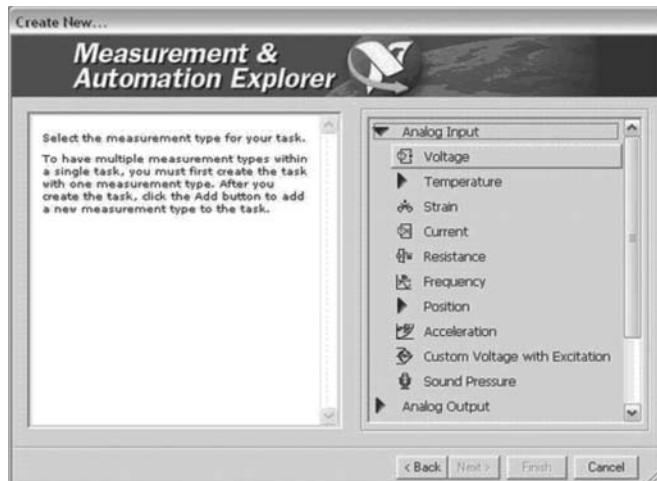


Figure 11.12 Analog input measurement.

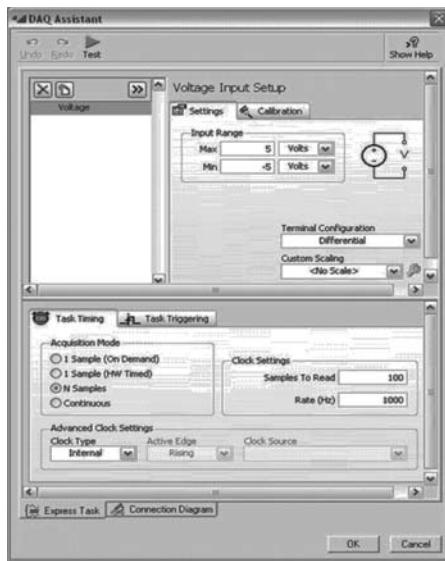
### 11.12.3 Configure the Task

After you create a task, you can configure channel-specific settings such as scaling, input limits and terminal configuration. You also can configure task-specific settings such as timing and triggering. In this task, you do not need to use scaling or triggering. You can configure settings like range (determines gain), terminal configuration, custom scaling and launch test panel. To configure the voltage measurement task, complete the following steps.

**Step 1:** Specify the input limits. You can use the default values of 5 for Max and -5 for Min if you do not know the theoretical limits for the signal you are measuring.

**Step 2:** Select the terminal configuration you used for the signal.

**Step 3:** On the *Task Timing* tab, select *Acquire N Samples*. Enter 100 for *Samples To Read* and enter 1000.00 for Rate (Hz) as shown in Figure 11.13.



**Figure 11.13** Configure the task.

#### 11.12.4 Test the Task

You can use test panels in the DAQ Assistant to test the task and make sure you connected the sensors properly. There is a test panel for each type of measurement. Complete the following steps to test the task.

**Step 1:** Launch the test panel for your task by clicking the *Test* button at the top of the screen.

**Step 2:** The test runs once automatically. Click the *Start* button to run the test again. Notice that the graph displays the acquired signal.

**Step 3:** Click the *Close* button when you are done. If necessary, modify the settings for the task and retest the task.

**Step 4:** After the test panel closes, click the *OK* button. The DAQ Assistant saves the voltage task, containing all the configuration information you entered, to MAX. You have created your voltage task.

#### 11.12.5 Generate LabVIEW Code

When you configure a task using the DAQ Assistant Express VI, the task is local to the application, and you cannot use it in other applications. You can convert a DAQ Assistant Express VI to a

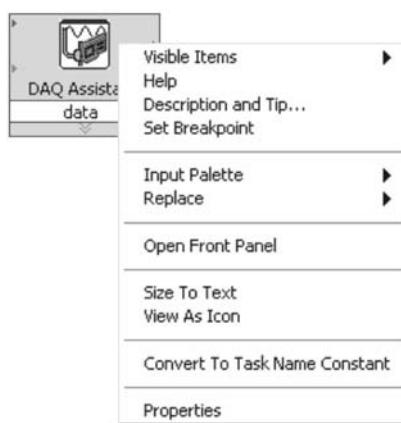
DAQmx Task Name control if you want to save the task to MAX and use it in other applications or to generate code. Complete the following steps to convert the DAQ Assistant Express VI to a DAQmx Task Name control.

**Step 1:** Right-click the DAQ Assistant Express VI and select *Convert to Task Name Constant* from the shortcut menu as shown in Figure 11.14.

**Step 2:** The DAQ Assistant launches, and you can modify the task, if necessary.

**Step 3:** Click the *OK* button.

**Step 4:** A DAQmx Task Name constant replaces the DAQ Assistant Express VI.



**Figure 11.14** DAQ Assistant Express VI.

You now have three options for generating code in LabVIEW from a task or channel:

- **Example**—It generates all the code necessary to run the task or channel, such as VIs needed to read or write samples, VIs to start and stop the task, loops and graphs. Choose this option if you want to run the task or channel you created to verify that it works or to use your configuration in a simple application. In LabVIEW, this option adds to the VI you are working in or creates a new VI. The generated code is a simple NI-DAQmx example that you can then modify for your application.
- **Configuration**—It generates the code that replicates the configuration of the tasks and channels. LabVIEW replaces the DAQmx Task Name control with a subVI that contains VIs and property nodes used for channel creation and configuration, timing configuration and triggering configuration used in the task or channel. Choose this option if you want to deploy your application to another system. Refer to *Deployment* in the *NI-DAQmx Help* located at *Start>Programs>National Instruments* for more information. When you generate Configuration code, the link between the application and the DAQ Assistant is lost. Any changes you make to the configuration code is not reflected in the DAQ Assistant. You can regenerate Configuration code from the DAQ Assistant, but the regenerated code does not incorporate previous changes that you made to the code.

- **Configuration and Example**—It generates both Configuration code and Example code for the task or channel in one step.

Complete the following steps to generate code to run the voltage task.

**Step 1:** Right-click the control on the front panel and select *Generate Code»Example* from the shortcut menu as in Figure 11.15.

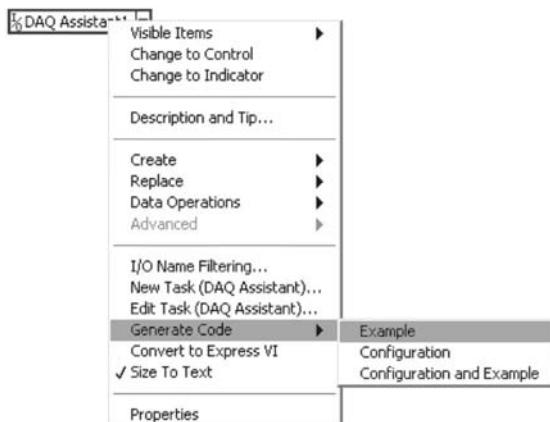


Figure 11.15 Generate code.

**Step 2:** View the block diagram. Notice that the DAQ Assistant generated all the code necessary to run the task, as shown in Figure 11.16. All of the timing information you set is contained in the DAQmx Task Name constant.

**Step 3:** Save the VI as MyVoltageTask.vi.

**Step 4:** View the front panel. Click the *Run* button to run the application.

**Note:** Any changes you make to the generated code apply only to the VI and are not saved into the task configuration stored in MAX.

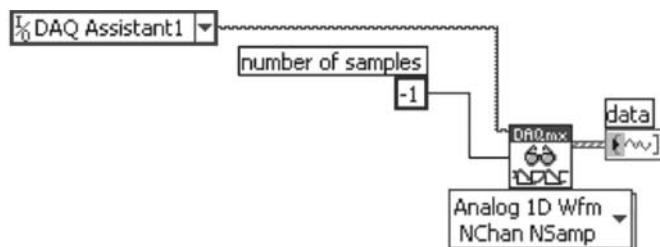


Figure 11.16 DAQ Assistant generated code.

### 11.13 CHANNELS AND TASK CONFIGURATION

In traditional NI-DAQ you can configure a set of virtual channels. Virtual channel is a collection of settings that can include a name, a physical channel, input terminal connections, the type of

measurement or generation, and scaling information. You can configure virtual channels with the DAQ Assistant, which you can open from Measurement & Automation Explorer (MAX) or your application software. You also can configure virtual channels with the NI-DAQmx API in your application program.

NI-DAQmx also includes tasks that are integral to the API. A task is a collection of one or more virtual channels with timing, triggering and other properties. Conceptually, a task represents a measurement or generation you want to perform. You can set up and save all of the configuration information in a task and use the task in an application. In NI-DAQmx, you can configure virtual channels as part of a task or separate from a task. Local channels are virtual channels created inside a task. Global channels are virtual channels defined outside a task. You can create global channels in MAX or in your application software. You can use global channels in any application or add them to a number of different tasks. If you modify a global channel, the change affects all tasks in which you reference that global channel. In most cases, it is simpler to use local channels.

## 11.14 SELECTING AND CONFIGURING A DATA ACQUISITION DEVICE

Depending on the application needs, you must determine the minimum number of analog input channels, analog output channels, and digital I/O lines that your data acquisition board requires. Other important factors to consider are the sampling rate, the input range, the input mode and the accuracy. When you connect any electrical signal to your data acquisition device, you expect your readings to match the electrical value of the input signal. Knowing that no measurement hardware is perfect, you must determine the best hardware configuration for improving your measurements.

One of the first things to consider is the field wiring. Depending on the type of signal source that you are using, you must configure the DAQ board in differential, nonreferenced single-ended or single-ended mode. In general, a differential measurement system is preferable because it rejects the ground loop-induced errors and the noise picked up in the environment to a certain degree. The single-ended configurations, on the other hand, provide twice as many measurement channels but are only appropriate if the magnitude of the induced errors is smaller than the required accuracy of the data.

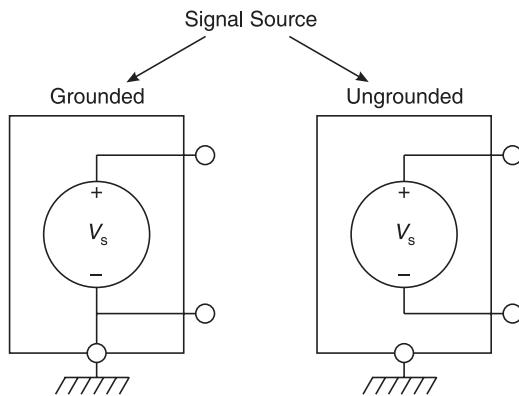
To get correct measurements you must properly ground your system. How the signal is grounded will affect how we ground the instrumentation amplifier on the DAQ device. Steps to proper grounding of your system are determine how your signal is grounded and choose a grounding mode for your measurement system.

### 11.14.1 Signal Sources

Analog input acquisitions use grounded and floating signal sources as in Figure 11.17. Signal sources can be grounded or ungrounded. The grounded signal sources are signals referenced to a system ground like earth ground and building ground. Because such sources use the system ground, they share a common ground with the measurement device. Examples are power supplies, signal generators and anything that plugs into an outlet ground. The grounds of two independently grounded signal sources generally are not at the same potential. The difference in ground potential between

two instruments connected to the same building ground system is typically from 10 mV to 200 mV. The difference can be higher if power distribution circuits are not properly connected. This causes a phenomenon known as a *ground loop*.

In a floating signal source, the voltage signal is not referenced to any common ground such as the earth or a building ground. Some common examples of floating signal sources are batteries, thermocouples, transformers and isolation amplifiers. Each terminal is independent of the system ground.



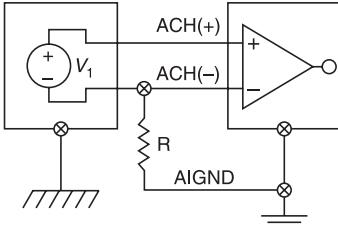
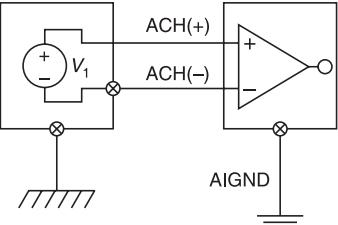
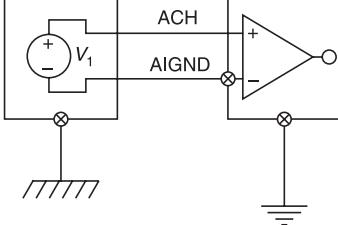
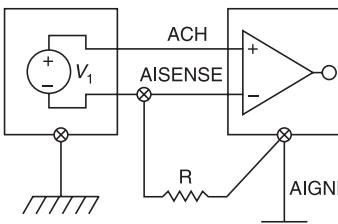
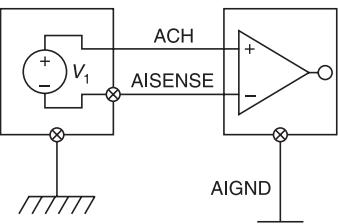
**Figure 11.17** Signal sources.

#### 11.14.2 Measurement Systems

You configure a measurement system based on the hardware you use and the measurement you take. Three modes of grounding for your Measurement System are Differential, Referenced Single-Ended (RSE) and Non-Referenced Single-Ended (NRSE). Mode you choose will depend on how your signal is grounded. Figure 11.18 shows the signal source and measurement systems summary.

Differential measurement systems are similar to floating signal sources in that you make the measurement with respect to a floating ground that is different from the measurement system ground. Neither of the inputs of a differential measurement system is tied to a fixed reference such as the earth or a building ground. Handheld, battery-powered instruments and DAQ devices with instrumentation amplifiers are examples of differential measurement systems.

Referenced and non-referenced single-ended measurement systems are similar to grounded sources in that you make the measurement with respect to a ground. A referenced single-ended measurement system measures voltage with respect to the ground, AIGND (analog input ground), which is directly connected to the measurement system ground. DAQ devices often use a non-referenced single-ended (NRSE) measurement technique, or pseudo differential measurement, which is a variant of the referenced single-ended measurement technique. In an NRSE measurement system, all measurements are still made with respect to a single-node analog input sense (AISENSE on E Series devices), but the potential at this node can vary with respect to the measurement system ground (AIGND). A single-channel NRSE measurement system is the same as a single-channel differential measurement system.

	Signal Source Type	
	Floating Signal Source (Not Connected to Building Ground)	Grounded Signal Source
Input	<p>Examples:</p> <ul style="list-style-type: none"> <li>• Ungrounded thermocouples</li> <li>• Signal conditioning with isolated outputs</li> <li>• Battery devices</li> </ul>	<p>Examples:</p> <ul style="list-style-type: none"> <li>• Plug-in instruments with nonisolated outputs</li> </ul>
Differential (DIFF)	 <p>See text for information on bias resistors.</p>	
Referenced Single-Ended (RSE)		<p>Not recommended</p> <p>Ground-loop losses, <math>V_g</math>, are added to measured signal.</p>
Non-Referenced Single-Ended (NRSE)	 <p>See text for information on bias resistors.</p>	

**Figure 11.18** Signal source and measurement systems summary.

### 11.14.3 Increasing Measurement Quality

When you design a measurement system, you may find that the measurement quality does not meet your expectations. You might want to record the smallest possible change in a voltage level. Perhaps you cannot tell if a signal is a triangle wave or a sawtooth wave and would like to see a better representation of the shape of a signal. Often, you want to reduce the noise in the signal. Methods for achieving these three increases in quality are necessary. The following reasons affect achieving the smallest detectable change in voltage:

- The resolution and range of the ADC
- The gain applied by the instrumentation amplifier
- The combination of the resolution, range and gain to calculate a property called the code width value.

*Resolution* is important. The number of bits used to represent an analog signal determines the resolution of the ADC. The resolution on a DAQ device is similar to the marks on a ruler. The more marks a ruler has, the more precise the measurements are. The higher the resolution is on a DAQ device, the higher the number of divisions into which a system can break down the ADC range, and therefore, the smaller the detectable change.

A 3-bit ADC divides the range into 23 or eight divisions. A binary or digital code between 000 and 111 represents each division. The ADC translates each measurement of the analog signal to one of the digital divisions. Figure 11.19 shows a 5-kHz sine wave digital image obtained by a 3-bit ADC. The digital signal does not represent the original signal adequately because the converter has too few digital divisions to represent the varying voltages of the analog signal. However, increasing the resolution to 16 bits to increase the ADC number of divisions from eight ( $2^3$ ) to 65,536 ( $2^{16}$ ) allows the 16-bit ADC to obtain an extremely accurate representation of the analog signal.

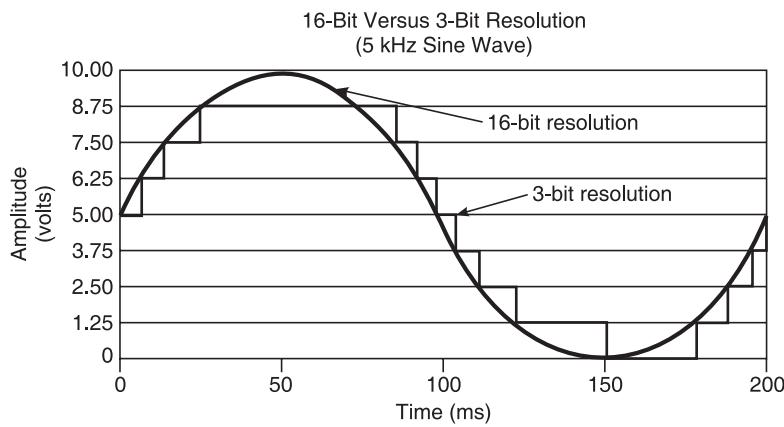
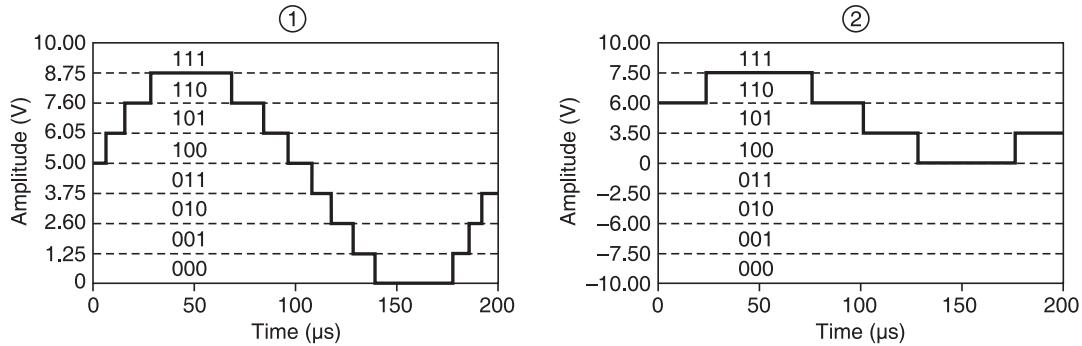


Figure 11.19 3-bit and 16-bit resolution.

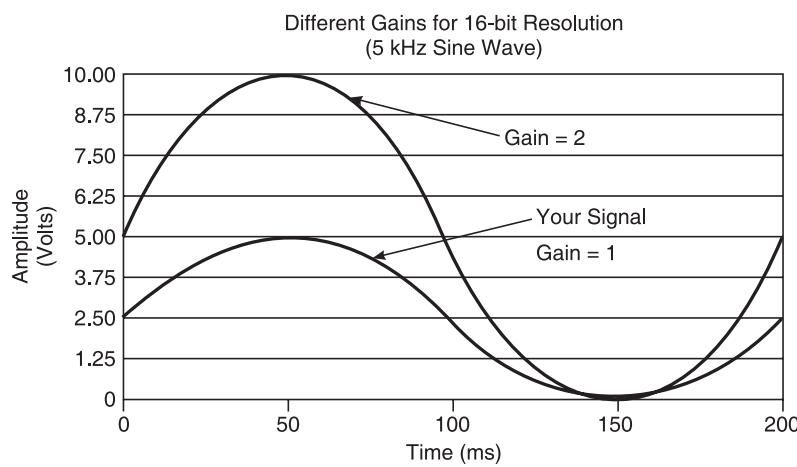
*Range* refers to the minimum and maximum analog signal levels that the ADC can digitize. Many DAQ devices feature selectable ranges (typically 0 to 10 V or -10 to 10 V), so you can match the ADC range to that of the signal to take best advantage of the available resolution to accurately measure the signal. For example, in Figure 11.20, the 3-bit ADC in chart 1 has eight digital divisions

in the range from 0 to 10 V, which is a unipolar range. If you select a range of -10 to 10 V, which is a bipolar range, as shown in chart 2, the same ADC separates a 20 V range into eight divisions. The smallest detectable voltage increases from 1.25 to 2.50 V, and the right chart is a much less accurate representation of the signal.



**Figure 11.20** Range example.

Amplification or attenuation of a signal can occur before the signal is digitized to improve the representation of the signal. By amplifying or attenuating a signal, you can effectively decrease the input range of an ADC and thus allow the ADC to use as many of the available digital divisions as possible to represent the signal. For example, Figure 11.21 shows the effects of applying amplification to a signal that fluctuates between 0 and 5 V using a 3-bit ADC and a range of 0 to 10 V. With no amplification, or gain = 1, the ADC uses only four of the eight divisions in the conversion. By amplifying the signal by two before digitizing, the ADC uses all eight digital divisions, and the digital representation is much more accurate. Effectively, the device has an allowable input range of 0 to 5 V because any signal above 5 V when amplified by a factor of two makes the input to the ADC greater than 10 V. The range, resolution, and amplification available on a DAQ device determine the smallest detectable change in the input voltage. This change in voltage represents one least significant bit (LSB) of the digital value and is also called the code width.



**Figure 11.21** Effects of applying amplification.

*Code width* is the smallest change in a signal that a system can detect. Code width is calculated using the following formula

$$C = D \left( \frac{1}{2^R} \right)$$

where  $C$  is code width,  $D$  is device input range, and  $R$  is bits of resolution.

Device input range is a combination of the gain applied to the signal and the input range of the ADC. For example, if the ADC input range is  $-10$  to  $+10$  V peak to peak and the gain is 2, the device input range is  $-5$  to  $+5$  V peak to peak, or 20 V. The smaller the code width, the more accurately a device can represent the signal. The formula confirms what you have already learned in the discussion on resolution, range and gain:

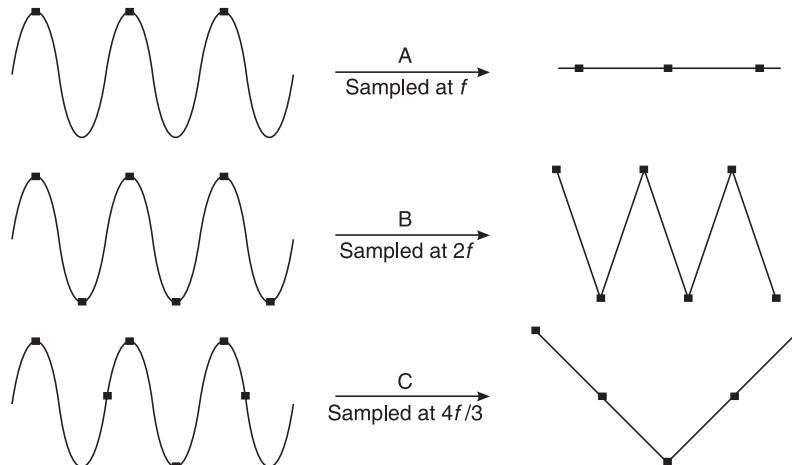
- Larger resolution = smaller code width = more accurate representation of the signal
- Larger amplification = smaller code width = more accurate representation of the signal
- Larger range = larger code width = less accurate representation of the signal

#### 11.14.4 Increasing Shape Recovery

The most effective way of increasing the shape recovery of a signal is to reduce your code width and increase your sampling frequency. To measure the frequency of your signal effectively, you must sample the signal at least the Nyquist frequency. The following states the Nyquist theorem:

$$f_{\text{sampling}} > 2(f_{\text{signal}})$$

where  $f_{\text{sampling}}$  is the sampling rate, and  $f_{\text{signal}}$  is the highest frequency component of interest in the measured signal. Figure 11.22 shows the effects of various sampling rates while sampling a signal.



**Figure 11.22** Effects of various sampling rates while sampling a signal.

The Nyquist theorem states that you must sample a signal at a rate greater than twice the highest frequency component of interest in the signal to capture the highest frequency component of interest. Otherwise, the high-frequency content aliases at a frequency inside the spectrum of interest, called the pass-band.

## 11.15 COMPONENTS OF COMPUTER BASED MEASUREMENT SYSTEM

Sensors and transducers detect physical phenomena. Signal conditioning components condition physical phenomena so that the measurement device can receive the data. The computer receives the data through the measurement device. Software controls the measurement system, telling the measurement device when and from which channels to acquire or generate data. Software also takes the raw data, analyzes it, and presents it in a form you can understand, such as a graph, chart, or file for a report.

NI measurement devices and application software are packaged with *NI-DAQ driver software* to program all the features of your NI measurement device such as configuring, acquiring and generating data from and sending data to NI measurement devices. Using NI-DAQ saves you from having to write these programs. *Application software*, such as LabVIEW, LabWindows/CVI, and Measurement Studio, sends the commands to the driver, such as acquire and return a thermocouple reading, and then displays and analyzes the data acquired. Figure 11.23 depicts the measurement system overview, showing the path of real-world physical phenomena to your measurement application.

- (i) **Real-world signals and physical phenomena**—Real-world signals and physical phenomena are signals like temperature, strain, vibration etc which are measured and controlled.
- (ii) **Sensors and transducers**—Sensors and transducers detect physical phenomena and provide voltage or current output to the signal conditioning devices.  
**Signal conditioning**—Signal conditioning devices condition physical phenomena which are detected by the sensors and transducers.
- (iii) **Data acquisition and modular instruments**—Data acquisition and modular instruments receive data from signal conditioning devices. The instruments are connected between the computer and signal conditioning devices. Data acquisition cards are placed inside the computer and modular instruments are placed outside the computer and connected through the cable.
- (iv) **Configuration using measurement and automation explorer (MAX)**—MAX is installed in the computer during the installation of the application software. MAX is used to configure the data acquisition devices and signal conditioning devices. MAX is also used to test the functioning of the DAQ devices and signal conditioning devices.
- (v) **Application programming interface (API) and driver engines**—NI-DAQ software is supplied with the data acquisition card. By installing the NI-DAQ software the drivers NI-DAQmx and Traditional NI-DAQ are installed in the computer. Both NI-DAQmx and Traditional NI-DAQ has their own application programming interface, hardware configuration and software configuration.  
**Traditional NI-DAQ**—Traditional NI-DAQ is an upgrade of the earlier version of NI-DAQ. It has the same VIs (built-in virtual instruments) and functions available in the previous versions. Both Traditional NI-DAQ and NI-DAQmx works on the same computer.

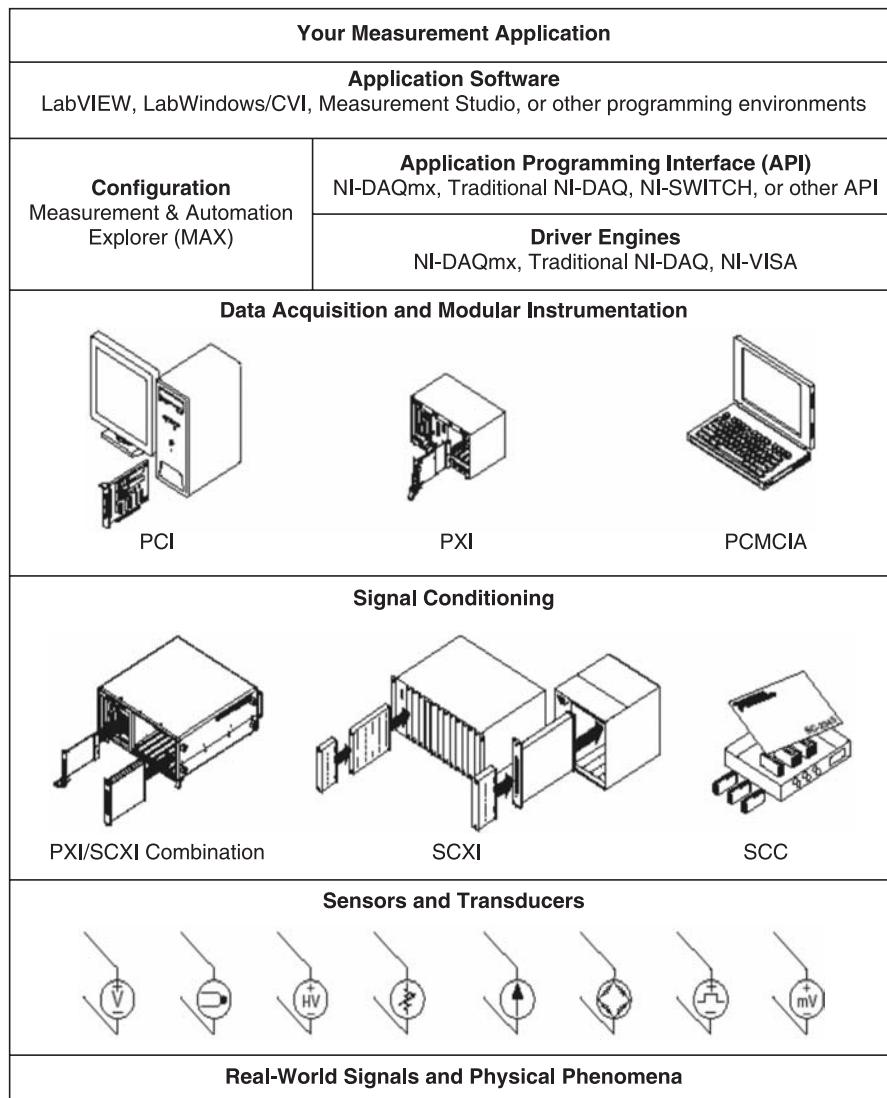


Figure 11.23 Measurement system overview.

**NI-DAQmx**—NI-DAQmx is the latest NI-DAQ driver with the following advantages over traditional NI-DAQ:

- DAQ assistant—a graphical way to configure channels and measurement tasks for the DAQ device and to generate NI-DAQmx code based on the channels and tasks, for use in LabVIEW, LabWindows/CVI, and Measurement Studio.
- Increased performance, including faster single-point analog I/O and multithreading.
- Simpler, more intuitive APIs for creating DAQ applications using fewer functions and VIs than earlier versions of NI-DAQ.

- Expanded functionality for LabVIEW, including property nodes and waveform data type support.
  - Similar APIs and functionality for ANSI C, LabWindows/CVI, and Measurement Studio, including native .NET and C++ interfaces.
  - Improved support and performance for the LabVIEW Real-Time module.
- (vi) **Application software**—National Instruments offers application software like LabVIEW, LabWindows, Measurement Studio etc. Here the application software used is LabVIEW. LabVIEW software is used to build application programmes based on the programmer's requirements. LabVIEW differs from other software by means of graphical programming. LabVIEW programs are called virtual instruments or VIs, because their appearance and operation imitate physical instruments, such as oscilloscopes and multimeters. LabVIEW contains comprehensive set of tools for acquiring, analyzing, displaying and sorting data as well as tools to help to troubleshoot the developed program.

#### 11.15.1 Installing the DAQ Card

Basic steps to install the DAQ card or device in a PCI device, PXI module, PCMCIA device or a USB/IEEE 1394 Devices are explained as follows.

**PCI devices**—Complete the following steps to install the PCI devices shown in Figure 11.24:

**Step 1:** Power off and unplug the computer.

**Step 2:** Remove the computer cover and/or the expansion slot cover.

**Step 3:** Touch any metal part of the computer to discharge any static electricity.

**Step 4:** Insert the device into a PCI system slot. Gently rock the device into place. Do *not* force the device into place.

**Step 5:** Secure the device mounting bracket to the computer back panel rail.

**Step 6:** Replace the computer cover, if applicable.

Steps to install the software are as follows:

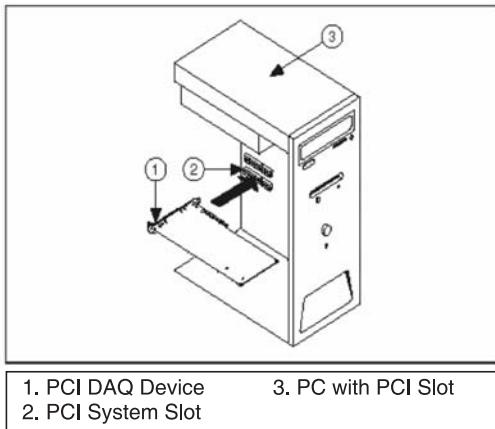
**Step 1:** Installing the DAQ card in the personal computer

**Step 2:** Installing the Application Software (LabVIEW)

**Step 3:** Installing the device drivers

**Step 4:** Configuration in the Measurement and Automation Explorer (MAX)

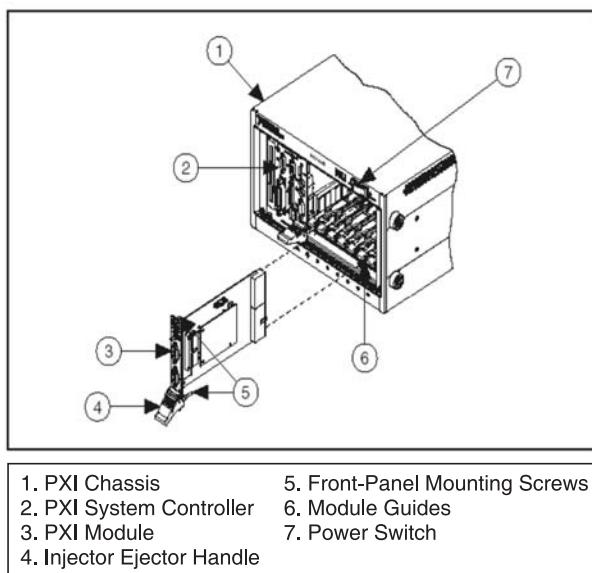
**Step 5:** *Signal conditioning devices, sensors and transducers*—Signals conditioning modules such as Signal Conditioning Extensions for Instrumentation (SCXI) are used to condition the physical signals acquired by the sensors and transducers. PCI-MIO-16E-4 DAQ card can be used. Signal Accessory Box is used for laboratory learning purposes instead of signal conditioning modules and transducers.



**Figure11.24** Installing the DAQ card.

**PXI devices**—Complete the following steps to install the PXI module as shown in Figure11.25:

- Step 1:** Power off and unplug the PXI chassis.
- Step 2:** Remove the filler panel of an unused PXI slot.
- Step 3:** Touch any metal part of the chassis to discharge static electricity.
- Step 4:** Ensure that the PXI module injector/ejector handle is not latched and swings freely.
- Step 5:** Place the PXI module edges into the module guides at the top and bottom of the chassis.



**Figure11.25** Installing the PXI module.

**Step 6:** Slide the device into the PXI slot to the rear of the chassis.

**Step 7:** When you begin to feel resistance, pull up on the injector/ejector handle to fully insert the device.

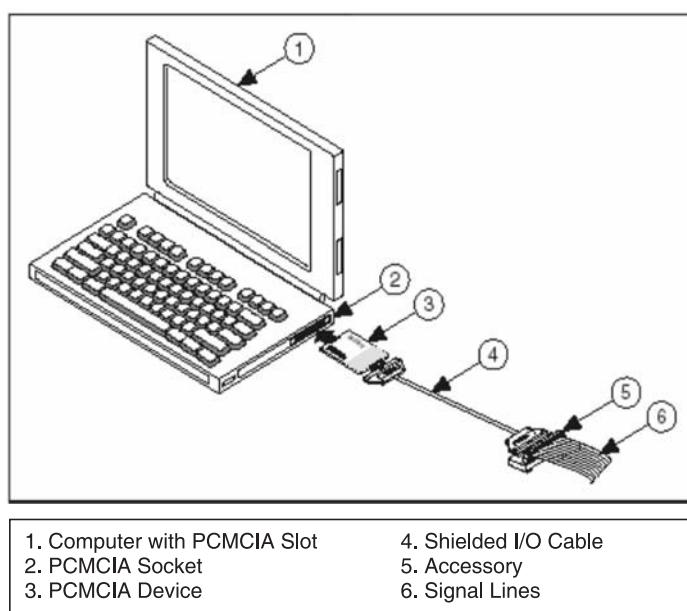
**Step 8:** Secure the device front panel to the chassis front panel mounting rail using the front-panel mounting screws.

**PCMCIA devices**—You can install the NI PCMCIA device in any available Type II PC Card slot. Complete the following steps to install the PCMCIA device as shown in Figure 11.26.

**Step 1:** Remove the PCMCIA slot cover on your computer, if any.

**Step 2:** Insert the PCMCIA bus connector of the PCMCIA device in the slot until the connector is firmly seated.

**Step 3:** Attach the I/O cable. Be careful not to put strain on the I/O cable when inserting or removing the cable connector. Always grasp the cable by the connector you are inserting or removing. *Never* pull directly on the I/O cable to unplug it from the PCMCIA device.



**Figure 11.26** Installing the NI PCMCIA device.

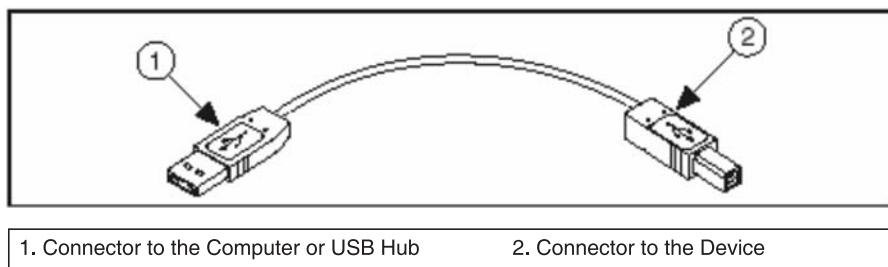
**USB/IEEE 1394 devices**—Complete the following steps to install an NI device for USB or IEEE 1394:

**Step 1:** Make power connections.

- If you are using the BP-1 battery pack, follow the installation instructions in your BP-1 installation guide.

- Some NI devices for USB or IEEE 1394 require external power.
  - ◆ If your device has an external power supply, verify that the voltage on the external power supply, if any, matches the voltage in your area (120 or 230 VAC) and the voltage required by your device. Connect one end of the power supply to an electrical outlet and the other end to your device.
  - ◆ If your device has a power cord, connect one end of the power cord to the device and the other end to an electrical outlet.

**Step 2:** Connect the cable from the computer USB or IEEE 1394 port or from any other hub or IEEE 1394 device to any available USB or IEEE 1394 port on the device. Figure 11.27 shows the USB cable and its connectors.



**Figure 11.27** USB cable and connectors.

**Step 3:** If you have a USB or IEEE 1394 device with a power switch, power on the device. The computer should immediately detect your device.

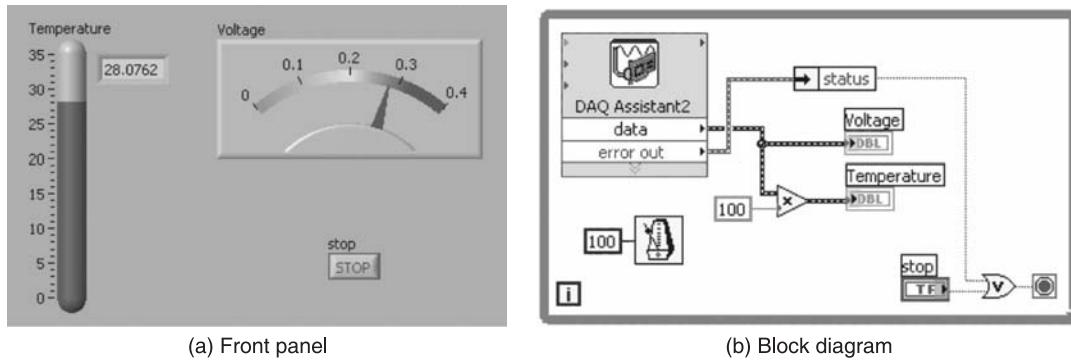
## SUMMARY

- The physical channel is a terminal or pin at which you can measure or generate an analog or digital signal.
- The DAQ Assistant is used to configure the DAQ device and perform data acquisition.
- Most applications can use the DAQ Assistant. For applications that require advanced timing and synchronization, use the VIs that come with NI-DAQmx.
- The DAQ Assistant can perform Analog Input, Analog Output, Digital I/O and Counter operations.
- LabVIEW communicates to your device through NI-DAQ.
- MAX is the primary configuration and testing utility that is available for the DAQ device.
- MAX has virtual channels, test panels, custom scales and software updates to help configure and test your system.
- The LabVIEW DAQ palettes contains all the functions necessary for your data acquisition program.
- DAQmx has one palette for all operations.
- Use property nodes to read and write settings for all DAQmx related options.

### MISCELLANEOUS SOLVED PROBLEMS

**Problem 11.1** Create a VI to acquire an analog signal (voltage output) of LM35 temperature sensor on the DAQ signal accessory. Using a scaling factor ( $v \times 100 = {}^\circ\text{C}$ ) convert the voltage to temperature and display both voltage and temperature values.

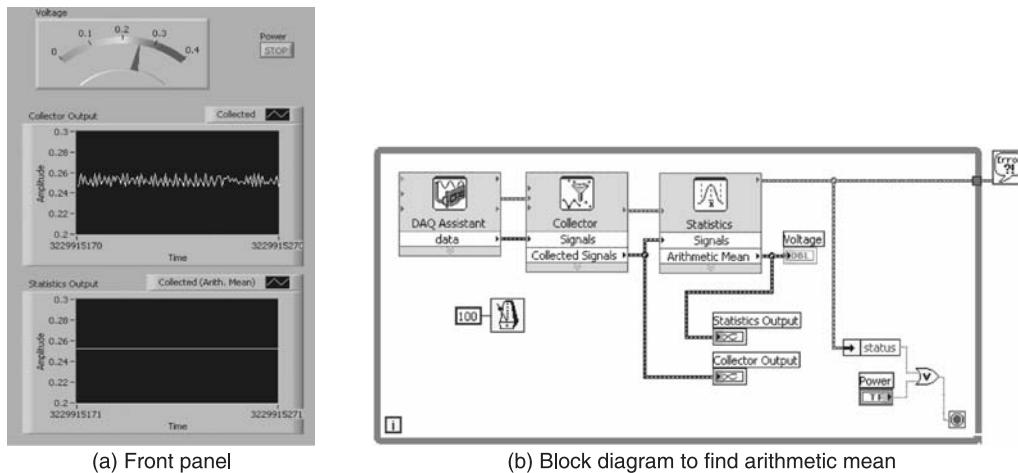
**Solution** The front panel and the block diagram to acquire an analog signal are shown in Figures P11.1(a) and P11.1(b).



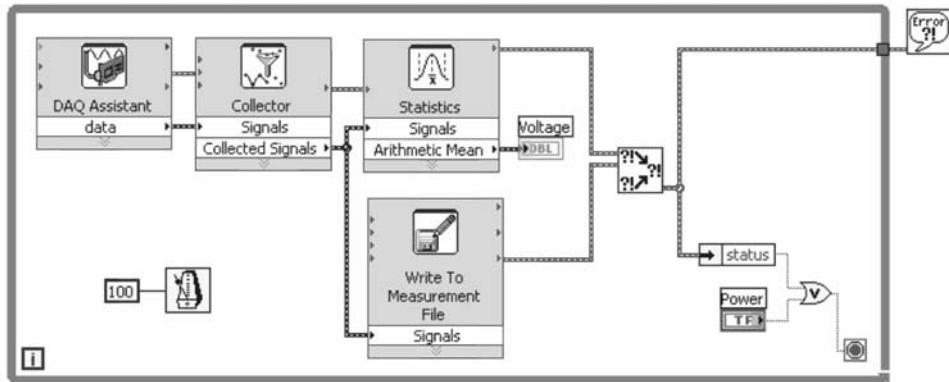
**Figure P11.1**

**Problem 11.2** Create a VI to read an analog input signal with noise through the data acquisition card and reduce noise in analog measurement by averaging. Find its arithmetic mean to average the signal. Plot both the acquired signal and averaged signal. Also write the acquired signal in a file using Write to Measurement File Express VI.

**Solution** The front panel is shown in Figure P11.2(a) and the block diagram to display the collected output and statistics output is shown in Figure P11.2(b), and the block diagram in Figure P11.2(c) uses the Write to Measurement File Express VI.



**Figure P11.2 (Contd.)**

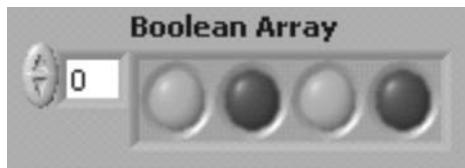


(c) Block diagram using Write to Measurement File Express VI

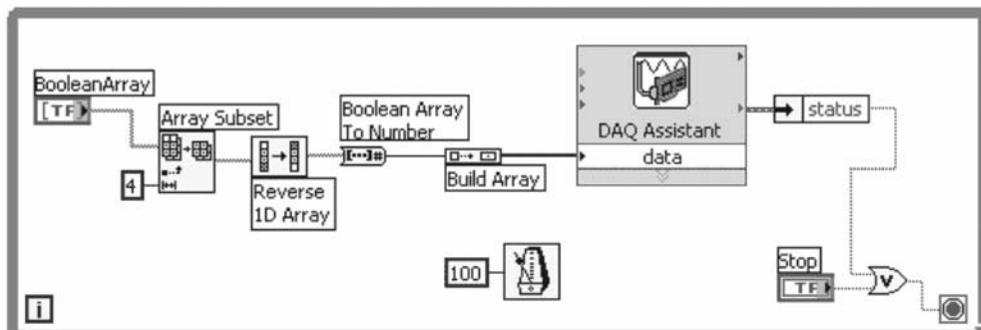
**Figure P11.2**

**Problem 11.3** Create a VI to produce four lines of digital outputs and to control the digital I/O lines on the DAQ device.

**Solution** The front panel and the block diagram to solve digital output problem are shown in Figures P11.3(a) and P11.3(b).



(a) Front panel

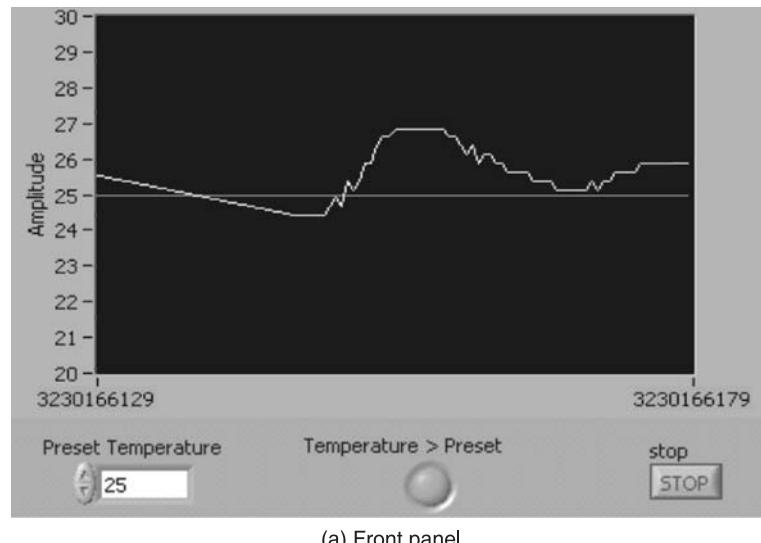


(b) Block diagram

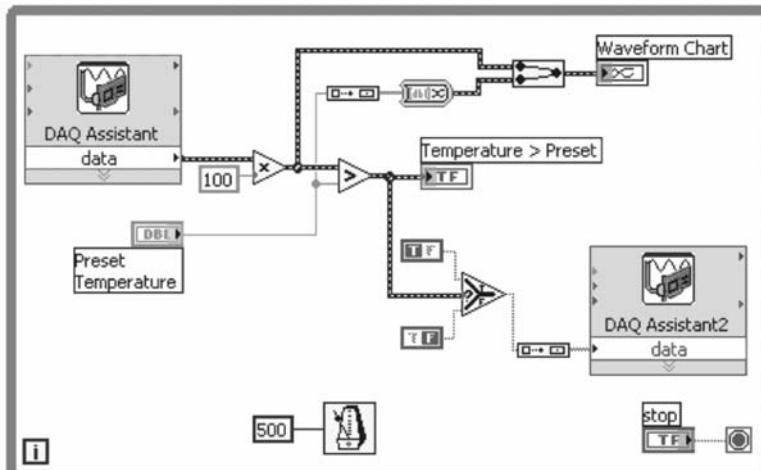
**Figure P11.3**

**Problem 11.4** Create a VI to read the temperature data from a temperature sensor as in Problem 11.1. Fix a preset temperature. Whenever the acquired temperature increases beyond the preset value, switch on a LED indication. Plot both acquired temperature and preset temperature in a single waveform chart.

**Solution** The front panel and the block diagram to read the temperature data from a temperature sensor are shown in Figures P11.4(a) and P11.4(b).



(a) Front panel



(b) Block diagram

**Figure P11.4**

**Problem 11.5** Create a VI to count an event using the counter terminals. It counts pulses from the quadrature encoder on the DAQ signal Accessory. When the count is equal to 10, switch ON an LED, otherwise switch OFF.

**Solution** The front panel and the block diagram to count an event using the counter terminals are shown in Figures P11.5(a) and P11.5(b).

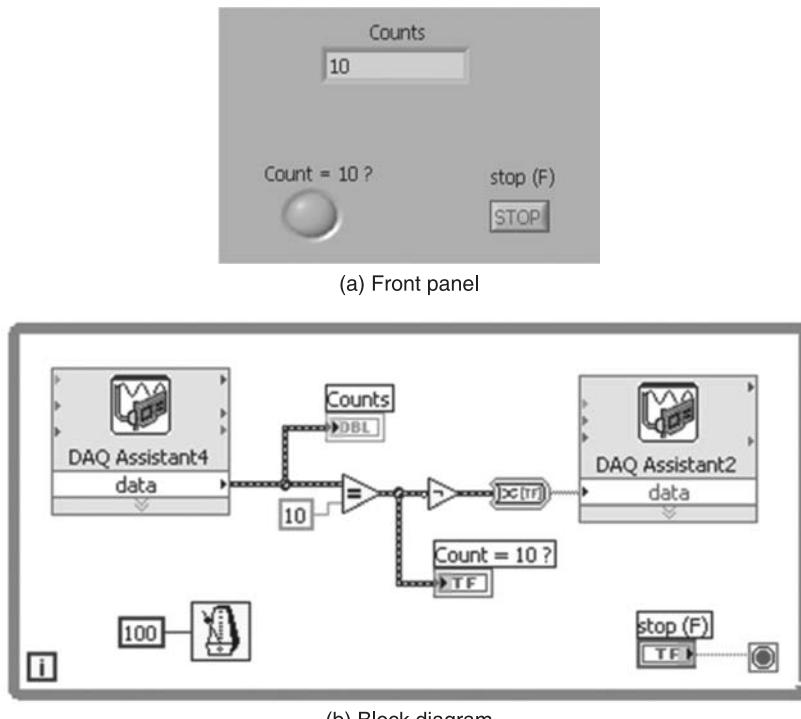


Figure P11.5

### REVIEW QUESTIONS

1. What are the major components of a PC-based data acquisition system?
2. Explain the types of transducers and the signal conditioning required for temperature measurement.
3. Explain how to perform an analog input. List the basic specifications for an analog input.
4. Describe the uses of Measurement & Automation Explorer.
5. Explain the significance of resolution, range and code width?
6. What is the role of DAQ software in a PC-based measurement system.
7. What is a plug-in DAQ device? Draw and explain the various functions in the DAQ device.
8. Explain the advantages of using the DAQ Assistant and list its main inputs and outputs.
9. What is a task in LabVIEW? Explain the method of creating and using a task.
10. Explain how acquired data can be stored and logged to disk.
11. Draw the block diagram and explain the measurement software framework.
12. What are the three methods that DAQ devices can be grounded?

## **EXERCISES**

---

1. Create a VI to produce voltage output from 0 to 10 volts in steps of 0.5 volts.
2. Create a VI to read the temperature data from a thermocouple by creating a lookup table of voltages and temperature equivalent.
3. Create a VI to count events using counter terminals of the data acquisition card.
4. Create a VI to satisfy the following control logic
  - (a) There are four pumps A,B,C and D and a water tank of height 0 meters.
  - (b) The pump A should be switched OFF if both B and C are simultaneously ON.
  - (c) The pump B should be switched OFF if both C and D are simultaneously ON.
  - (d) The pump C is switched ON if the temperature of the tank exceeds 27°C.
  - (e) The pump D should be switched ON only if the tank level is less than 10 meters.
  - (f) Use the digital port to indicate the pump running condition and use the analog input channel to measure the tank temperature.



# IMAQ VISION

---

## 12.1 VISION BASICS

Vision Basics contains the information about digital images. It contains information about the properties of digital images, image types, file formats, the internal representation of images in IMAQ Vision, image borders and image masks. The next important aspect is the display. It contains information about image display, palettes, regions of interest and nondestructive overlays. The last topic is about system setup and calibration. It describes how to set up an imaging system and calibrate the imaging setup so that you can convert pixel coordinates to real-world coordinates.

### 12.1.1 Digital Images

Digital Images contain information about the properties of digital images, image types, file formats, the internal representation of images in IMAQ Vision, image borders and image masks. An image is a 2D array of values representing light intensity. For the purposes of image processing, the term image refers to a digital image. An image is a function of the light intensity  $f(x, y)$  where  $f$  is the brightness of the point  $(x, y)$ , and  $x$  and  $y$  represent the spatial coordinates of a picture element or pixel.

In digital image processing, an imaging sensor converts an image into a discrete number of pixels. The imaging sensor assigns to each pixel a numeric location and a gray level or color value that specifies the brightness or color of the pixel. A digitized image has three basic properties: resolution, definition and number of planes.

Image resolution is the spatial resolution of an image is determined by its number of rows and columns of pixels. An image composed of  $m$  columns and  $n$  rows has a resolution of  $m \times n$ . This image has  $m$  pixels along its horizontal axis and  $n$  pixels along its vertical axis. The definition of an image indicates the number of shades that you can see in the image. The number of planes in an image corresponds to the number of arrays of pixels that compose the image. A grayscale or pseudo-color image is composed of one plane, while a true-color image is composed of three planes—one each for the red component, blue component and green component.

Image types in the IMAQ Vision libraries can manipulate three types of images: grayscale, color and complex images. A grayscale image is composed of a single plane of pixels. Each pixel is encoded using one of the following single numbers:

- An 8-bit unsigned integer representing grayscale values between 0 and 255.
- A 16-bit signed integer representing grayscale values between -32,768 and +32,767.
- A single-precision floating point number, encoded using four bytes, representing grayscale values ranging from  $-\infty$  to  $\infty$ .

A color image is encoded in memory as either a red, green and blue (RGB) image or a hue, saturation, and luminance (HSL) image. Color image pixels are a composite of four values. RGB images store color information using 8 bits each for the red, green and blue planes. HSL images store color information using 8 bits each for hue, saturation and luminance.

A complex image contains the frequency information of a grayscale image. You can create a complex image by applying a Fast Fourier Transform (FFT) to a grayscale image. After you transform a grayscale image into a complex image, you can perform frequency domain operations on the image.

An image file is composed of a header followed by pixel values. Depending on the file format, the header contains image information about the horizontal and vertical resolution, pixel definition, and the original palette. Image files may also store information about calibration, pattern matching templates, and overlays. The following are common image file formats:

- Bitmap (BMP)
- Tagged image file format (TIFF)
- Portable network graphics (PNG)—Offers the capability of storing image information about spatial calibration, pattern matching templates and overlays.
- Joint Photographic Experts Group format (JPEG)
- National Instruments internal image file format (AIPD)—Used for saving floating-point, complex and HSL images.

Standard formats for 8-bit grayscale and RGB color images are BMP, TIFF, PNG, JPEG and AIPD. Standard formats for 16-bit grayscale, 64-bit RGB and complex images are PNG and AIPD.

Many image processing functions process a pixel by using the values of its neighbors. A *neighbor* is a pixel whose value affects the value of a nearby pixel when an image is processed. Pixels along the edge of an image do not have neighbors on all four sides. If you need to use a function that processes pixels based on the value of their neighboring pixels, specify an *imageborder* that surrounds the image to account for these outlying pixels. An *image mask* isolates parts of an image for processing. If a function has an image mask parameter, the function process or analysis depends on both the source image and the image mask.

### 12.1.2 Display

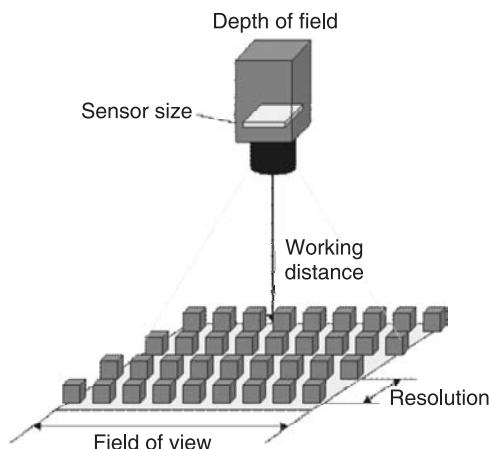
Displaying images is an important component of a vision application because it gives you the ability to visualize your data. *Image processing* and *image visualization* are distinct and separate elements. Image processing refers to the creation, acquisition and analysis of images. Image visualization refers to how image data is presented and how you can interact with the visualized images. A typical imaging application uses many images in memory that the application never displays.

One of the key components of displaying images is the display mode that the video adaptor operates. The display mode indicates how many bits specify the color of a pixel on the display screen. Generally, the display mode available from a video adaptor ranges from 8 bits to 32 bits per pixel, depending on the amount of video memory available on the video adaptor and the screen resolution you choose.

At the time a grayscale image is displayed on the screen, IMAQ Vision converts the value of each pixel of the image into red, green and blue intensities for the corresponding pixel displayed on the screen. This process uses a color table, called a *palette*, which associates a color to each possible grayscale value of an image. IMAQ Vision provides the capability to customize the palette used to display an 8-bit grayscale image. A palette is a pre-defined or user-defined array of RGB values.

### 12.1.3 System Setup and Calibration

Before you acquire, analyze and process images, you must set up your imaging system. Five factors comprise a imaging system: field of view, working distance, resolution, depth of field, and sensor size. Figure 12.1 illustrates these concepts.



**Figure 12.1** Fundamental parameters of an imaging system.

- **Resolution**—The smallest feature size on your object that the imaging system can distinguish.
- **Pixel resolution**—The minimum number of pixels needed to represent the object under inspection.

- **Field of view**—The area of the object under inspection that the camera can acquire.
- **Working distance**—The distance from the front of the camera lens to the object under inspection.
- **Sensor size**—The size of a sensor's active area, typically defined by the sensor's horizontal dimension.
- **Depth of field**—The maximum object depth that remains in focus.

Spatial calibration is the process of computing pixel to real-world unit transformations while accounting for many errors inherent to the imaging setup. Calibrating your imaging setup is important when you need to make accurate measurements in real-world units.

IMAQ Vision has two algorithms for calibration: perspective and nonlinear. Perspective calibration corrects for perspective errors, and nonlinear calibration corrects for perspective errors and nonlinear lens distortion. Learning for perspective is faster than learning for nonlinear distortion. The perspective algorithm computes one pixel to real-world mapping for the entire image. You can use this mapping to convert the coordinates of any pixel in the image to real-world units. The nonlinear algorithm computes pixel to real-world mappings in a rectangular region centered around each dot in the calibration grid.

## 12.2 IMAGE PROCESSING AND ANALYSIS

This section also describes conceptual information about image analysis and processing, operators and frequency domain analysis. Image processing and analysis contains image analysis with information about histograms, line profiles, and intensity measurements. Image processing provides information about lookup tables, kernels, spatial filtering and grayscale morphology. Operators contain information about arithmetic and logic operators that mask, combine and compare images. Frequency domain analysis contains information about frequency domain analysis, the fast Fourier transform, and analyzing and processing images in the frequency domain.

### 12.2.1 Image Analysis

Image analysis combines techniques that compute statistics and measurements based on the gray-level intensities of the image pixels. You can use the image analysis functions to understand the content of the image and to decide which type of inspection tools to use to solve your application. Image analysis functions also provide measurements that you can use to perform basic inspection tasks such as presence or absence verification.

A histogram counts and graphs the total number of pixels at each grayscale level. From the graph, you can tell whether the image contains distinct regions of a certain gray-level value. A histogram provides a general description of the appearance of an image and helps identify various components such as the background, objects and noise. The histogram is a fundamental image analysis tool that describes the distribution of the pixel intensities in an image. Use the histogram to determine if the overall intensity in the image is high enough for your inspection task. You can use the histogram to determine whether an image contains distinct regions of certain grayscale values. You also can use a histogram to adjust the image acquisition conditions.

You can detect two important criteria by looking at the histogram. They are saturation and lack of contrast. In saturation too little light in the imaging environment leads to underexposure of

the imaging sensor, while too much light causes overexposure, or saturation, of the imaging sensor. Images acquired under underexposed or saturated conditions will not contain all the information that you want to inspect from the scene being observed. It is important to detect these imaging conditions and correct for them during setup of your imaging system. You can detect whether a sensor is underexposed or saturated by looking at the histogram. An underexposed image contains a large number of pixels with low gray-level values. This appears as a peak at the lower end of the histogram. An overexposed or saturated image contains a large number of pixels with very high gray-level values.

Lack of contrast is a widely-used type of imaging application and involves inspecting and counting parts of interest in a scene. A strategy to separate the objects from the background relies on a difference in the intensities of both, for example, a bright part and a darker background. In this case, the analysis of the histogram of the image reveals two or more well-separated intensity populations.

Tune your imaging setup until the histogram of your acquired images has the contrast required by your application. Two types of histograms can be calculated: the linear and cumulative histograms. The vertical axis of a histogram plot can be shown in a linear or logarithmic scale.

A line profile plots the variations of intensity along a line. It returns the grayscale values of the pixels along a line and graphs it. The peaks and valleys represent increases and decreases of the light intensity along the line selected in the image. Their width and magnitude are proportional to the size and intensity of their related regions. For example, a bright object with uniform intensity appears in the plot as a plateau. The higher is the contrast between an object and its surrounding background, the steeper is the slopes of the plateau. Noisy pixels, on the other hand, produce a series of narrow peaks. Intensity measurements measure the grayscale image statistics in an image or regions in an image.

### 12.2.2 Image Processing

Image processing contains information about lookup tables, convolution kernels, spatial filters and grayscale morphology. The lookup table (LUT) transformations are basic image-processing functions that highlight details in areas containing significant information at the expense of other areas. These functions include histogram equalization, gamma corrections logarithmic corrections, and exponential corrections. Use LUT transformations to improve the contrast and brightness of an image by modifying the dynamic intensity of regions with poor contrast. A LUT transformation converts input gray-level values from the source image into other gray-level values in the transformed image.

Seven predefined LUTs are available in IMAQ Vision: Linear, Logarithmic, Power 1/Y, Square Root, Exponential, Power Y and Square. A convolution kernel defines a 2D filter that you can apply to a grayscale image. A convolution kernel is a 2D structure whose coefficients define the characteristics of the convolution filter that it represents. In a typical filtering operation, the coefficients of the convolution kernel determine the filtered value of each pixel in the image. IMAQ Vision provides a set of convolution kernels that you can use to perform different types of filtering operations on an image. You also can define your own convolution kernels, thus creating custom filters. Use a convolution kernel whenever you want to filter a grayscale image.

Filtering a grayscale image enhances the quality of the image to meet the requirements of your application. Use filters to smooth an image, remove noise from an image, enhance the edge information in an image, and so on. Spatial Filtering filters are divided into two types: linear (also called convolution) and nonlinear. A convolution is an algorithm that consists of recalculating the value of a pixel based on its own pixel value and the pixel values of its neighbors weighted by the coefficients of a convolution kernel. The sum of this calculation is divided by the sum of the elements in the kernel to obtain a new pixel value.

Grayscale morphology is morphological transformations extract and alter the structure of particles in an image. They fall into two categories:

- Binary morphology functions which apply to binary images
- Grayscale morphology functions which apply to gray-level images

In grayscale morphology, a pixel is compared to those pixels surrounding it in order to keep the pixels whose values are the smallest (in the case of an erosion) or the largest (in the case of a dilation). Use grayscale morphology functions to filter or smooth the pixel intensities of an image. Applications include noise filtering, uneven background correction, and gray-level feature extraction. Gray-level morphology functions are erosion, dilation, opening, closing, proper-opening, proper-closing and auto-median.

### 12.2.3 Operators

Arithmetic and logic operators can mask, combine and compare images. Operators perform basic arithmetic and logical operations on images. Use operators to add, subtract, multiply and divide an image with other images or constants. You also can perform logical operations, such as AND/NAND, OR/NOR, and XOR/XNOR, and make pixel comparisons between an image and other images or a constant. Common applications of these operators include time-delayed comparisons, identification of the union or intersection between images, correction of image backgrounds to eliminate light drifts, and comparisons between several images and a model. You also can use operators to threshold or mask images and to alter contrast and brightness.

### 12.2.4 Frequency Domain Analysis

Information about converting images into the frequency domain using the fast Fourier transform, and information about analyzing and processing images in the frequency domain are important. Frequency filters alter pixel values with respect to the periodicity and spatial distribution of the variations in light intensity in the image. Unlike spatial filters, frequency filters do not apply directly to a spatial image, but to its frequency representation. The frequency representation of an image is obtained through the fast Fourier transform (FFT) function which reveals information about the periodicity and dispersion of the patterns found in the source image.

You can filter the spatial frequencies seen in an FFT image. The inverse FFT function then restores a spatial representation of the filtered FFT image. Frequency processing is another technique for extracting information from an image. Instead of using the location and direction of light-intensity variations, you can use frequency processing to manipulate the frequency of the occurrence of these variations in the spatial domain. This new component is called the *spatial frequency* which is the frequency with which the light intensity in an image varies as a function of spatial coordinates.

Use a lowpass frequency filter to attenuate or remove, or truncate, high frequencies present in the image. This filter suppresses information related to rapid variations of light intensities in the spatial image. An inverse FFT, used after a lowpass frequency filter, produces an image in which noise, details, texture, and sharp edges are smoothed. A highpass frequency filter attenuates or removes, or truncates, low frequencies present in the complex image. This filter suppresses information related to slow variations of light intensities in the spatial image. In this case, an inverse FFT used after a highpass frequency filter produces an image in which overall patterns are sharpened and details are emphasized. A mask frequency filter removes frequencies contained in a mask specified by the user. Using a mask to alter the Fourier transform of an image offers more possibilities than applying a lowpass or highpass filter. The image mask is composed by the user and can describe very specific frequencies and directions in the image. You can apply this technique, for example, to filter dominant frequencies as well as their harmonics in the frequency domain.

## 12.3 PARTICLE ANALYSIS

Conceptual information about particle analysis, including thresholding, morphology and particle measurements need to be understood. Thresholding provides information about thresholding and color thresholding. Binary morphology gives information about structuring elements, connectivity, and primary and advanced morphological transformations. Particle measurements is about characterizing digital particles.

You can use particle analysis to detect connected regions or groupings of pixels in an image and then make selected measurements of those regions. These regions are commonly referred to as *particles*. A particle is a contiguous region of nonzero pixels. You can extract particles from a grayscale image by thresholding the image into background and foreground states. Zero-valued pixels are in the background state, and all nonzero valued pixels are in the foreground state.

Particle analysis consists of a series of processing operations and analysis functions that produce information about particles in an image. Using particle analysis, you can detect and analyze any 2D shape in an image. Use particle analysis when you are interested in finding particles whose spatial characteristics satisfy certain criteria. In many applications where computation is time-consuming, you can use particle filtering to eliminate particles that are of no interest based on their spatial characteristics, and keep only the relevant particles for further analysis. You can use particle analysis to find statistical information—such as the presence of particles, their number and size, and location. This information allows you to perform many machine vision inspection tasks— such as detecting flaws on silicon wafers, detecting soldering defects on electronic boards, or web inspection applications such as finding structural defects on wood planks or detecting cracks on plastics sheets.

You also can locate objects in motion control applications. In applications where there is a significant variance in the shape or orientation of an object, particle analysis is a powerful and flexible way to search for the object. You can use a combination of the measurements obtained through particle analysis to define a feature set that uniquely defines the shape of the object.

### 12.3.1 Thresholding

Thresholding consists of segmenting an image into two regions: a particle region and a background region. This process works by setting to 1 all pixels that belong to a gray-level interval, called the

threshold interval, and setting all other pixels in the image to 0. Use thresholding to isolate objects of interest in an image. Thresholding converts the image from a grayscale image, with pixel values ranging from 0 to 255, to a binary image with pixel values of 0 or 1.

Thresholding enables you to select ranges of pixel values in grayscale and color images that separate the objects under consideration from the background. Pixels outside the threshold interval are set to 0 and are considered as part of the background area. Pixels inside the threshold interval are set to 1 and are considered as part of a particle area.

Particles are characterized by an *intensity range*. They are composed of pixels with gray-level values belonging to a given threshold interval (overall luminosity or gray shade). All other pixels are considered to be part of the background. All automatic thresholding methods use the histogram of an image to determine the threshold. IMAQ Vision has five automatic thresholding techniques like Clustering, Entropy, InterVariance, Metric and Moments.

Color thresholding converts a color image into a binary image. Threshold a color image when you need to isolate features for analysis and processing or to remove unnecessary features. To threshold a color image, specify a threshold interval for each of the three-color components. A pixel in the output image is set to 1 if and only if its color components fall within the specified ranges. Otherwise, the pixel value is set to 0.

### 12.3.2 Binary Morphology

*Binary morphological operations* extract and alter the structure of particles in a binary image. You can use these operations during your inspection application to improve the information in a binary image before making particle measurements such as the area, perimeter and orientation. A *binary image* is an image containing particle regions with pixel values of 1 and a background region with pixel values of 0. Binary images are the result of the *thresholding* process. Because thresholding is a subjective process, the resulting binary image may contain unwanted information such as noise particles, particles touching the border of images, particles touching each other, and particles with uneven borders. By affecting the shape of particles, morphological functions can remove this unwanted information, thus improving the information in the binary image.

Morphological operators that change the shape of particles process a pixel based on its number of neighbors and the values of those neighbors. A *neighbor* is a pixel whose value affects the values of nearby pixels during certain image processing functions. Morphological transformations use a 2D binary mask called a *structuring element* to define the size and effect of the neighborhood on each pixel, controlling the effect of the binary morphological functions on the shape and the boundary of a particle.

Use a structuring element when you perform any primary binary morphology operation or the advanced binary morphology operation Separation. You can modify the size and the values of a structuring element to alter the shape of particles in a specific way. However, study the basic morphology operations before defining your own structuring element.

### 12.3.3 Particle Measurements

A particle is a group of contiguous nonzero pixels in an image. Particles can be characterized by measurements related to their attributes such as particle location, area and shape. Use particle measurements when you want to make shape measurements on particles in a binary image. In

addition to making conventional pixel measurements, IMAQ Vision particle analysis functions can use calibration information attached to an image to make measurements in calibrated real-world units. In applications that do not require the display of corrected images, you can use the calibration information attached to the image to make real-world measurements directly without using time-consuming image correction. In pixel measurements, a pixel is considered to have an area of one square unit, located entirely at the center of the pixel. In calibrated measurements, a pixel is a polygon with corners defined as plus or minus one half a unit from the center of the pixel.

## 12.4 MACHINE VISION

IMAQ product family consists of IMAQ Hardware, NI-IMAQ and IMAQ Vision. Most IMAQ devices work with motion control and DAQ hardware using the real-time system intergration (RTSI) bus. They perform tasks as pixel and line scaling (decimation) and region of interest acquisition. NI-IMAQ features a complete and robust API for image acquisition. It features an extensive library of functions that you can call from application programming environment. These functions include routines for video configuration, image acquisition (continuous and single-shot), memory buffer allocation, trigger control and board configuration. NI-IMAQ provides all the functionality you need to acquire and save images. IMAQ vision in the image processing toolkit or library adds high-level machine vision and image processing to your programming environment. You must have LabVIEW, Measurement Studio, Visual C++, Visual Basic, or another programming environment to use IMAQ Vision. Use IMAQ Vision to accelerate the development of industrial machine vision and scientific image applications. IMAQ Vision builder is a tool for prototyping and testing image processing applications. IMAQ Vision builder uses the IMAQ Vision library but is a stand alone executable that you can use independently of other programs.

Machine Vision describes conceptual information about high-level operations commonly used in machine vision applications such as edge detection, pattern matching, dimensional measurements, color inspection, binary particle classification, optical character recognition and instrument reading:

*Edge Detection* contains information about edge detection techniques and tools that locate edges, such as the rake, concentric rake, spoke and caliper.

*Pattern Matching* contains information about pattern matching.

*Geometric Matching* contains information about geometric matching and when to use it instead of pattern matching.

*Dimensional Measurements* contains information about analytic tools, clamps, line fitting, and coordinate systems.

*Color Inspection* contains information about color spaces, the color spectrum, color matching, color location, and color pattern matching.

*Binary Particle Classification* contains information about training and classifying objects in an image.

*Optical Character Recognition* contains information about training and reading text and/or characters in an image.

*Instrument Readers* contains information about reading meters, LCDs, and barcodes.

### 12.4.1 Edge Detection

*Edge detection* finds edges along a line of pixels in the image. Use the edge detection tools to identify and locate discontinuities in the pixel intensities of an image. The discontinuities are typically associated with abrupt changes in pixel intensity values that characterize the boundaries of objects in a scene. To detect edges in an image, specify a search region in which to locate images. You can specify the search region interactively or programmatically. When specified interactively, you can use one of the line ROI tools to select the search path you want to analyze. You also can programmatically fix the search regions based either on constant values or the result of a previous processing step. For example, you may want to locate edges along a specific portion of a part that has been previously located using particle analysis or pattern matching algorithms. The edge detection software analyzes the pixels along this region to detect edges. You can configure the edge detection tool to find all edges, find the first edge, or find the first and last edges in the region. Edge detection is an effective tool for many machine vision applications. It provides your application with information about the location of object boundaries and the presence of discontinuities. An *edge* is a significant change in the grayscale values between adjacent pixels in an image. In IMAQ Vision, edge detection works on a 1D profile of pixel values along a search region. Use edge detection in the following three application areas: gauging, detection and alignment.

*Gauging* applications are used to make critical dimensional measurements—such as length, distance, diameter, angle and quantity—to determine if the product under inspection is manufactured correctly. Depending on whether the gauged parameters fall inside or outside of the user-defined tolerance limits, the component or part is either classified or rejected. Gauging is often used both inline and offline in production. During inline processes, each component is inspected as it is manufactured. Visual inline gauging inspection is a widely used inspection technique in applications such as mechanical assembly verification, electronic packaging inspection, container inspection, glass vial inspection and electronic connector inspection. Similarly, gauging applications often measure the quality of products offline. First, a sample of products is extracted from the production line. Next, measured distances between features on the object are studied to determine if the sample falls within a tolerance range. You can measure the distances separating the different edges located in an image, as well as positions measured using particle analysis or pattern matching techniques. Edges can also be combined in order to derive best fit lines, projections, intersections and angles. Use edge locations to compute estimations of shape measurements such as circles, ellipses, polygons and so on.

*Detection* applications are typical in electronic connector assembly and mechanical assembly applications. The objective of the application is to determine if a part is present or not present using line profiles and edge detection. An edge along the line profile is defined by the level of contrast between background and foreground and the slope of the transition. Using this technique, you can count the number of edges along the line profile and compare the result to an expected number of edges. This method offers a less numerically intensive alternative to other image processing methods such as image correlation and pattern matching. Use edge detection to detect structural defects, such as cracks, or cosmetic defects such as scratches, on a part. If the part is of uniform intensity, these defects show up as sharp changes in the intensity profile. Edge detection identifies these changes.

*Alignment* determines the position and orientation of a part. In many machine vision applications, the object that you want to inspect may be at different locations in the image. Edge

detection finds the location of the object in the image before you perform the inspection, so that you can inspect only the regions of interest. The position and orientation of the part can also be used to provide feedback information to a positioning device such as a stage.

#### 12.4.2 Pattern Matching

*Pattern matching* quickly locates regions of a grayscale image that match a known reference pattern, also referred to as a model or *template*. When using pattern matching, you create a template that represents the object for which you are searching. Your machine vision application then searches for instances of the template in each acquired image, calculating a score for each match. This score relates how closely the template resembles the located matches. Pattern matching finds template matches regardless of lighting variation, blur, noise, and geometric transformations such as shifting, rotation, or scaling of the template.

Pattern matching algorithms are some of the most important functions in machine vision because of their use in varying applications. You can use pattern matching in the following three general applications:

- **Alignment**—Determines the position and orientation of a known object by locating *fiducials*. Use the fiducials as points of reference on the object.
- **Gauging**—Measures lengths, diameters, angles and other critical dimensions. If the measurements fall outside set tolerance levels, the component is rejected. Use pattern matching to locate the object you want to gauge.
- **Inspection**—Detects simple flaws such as missing parts or unreadable print.

Pattern matching provides your application with the number of instances and the locations of template matches within an inspection image. For example, you can search an image containing a printed circuit board (PCB) for one or more fiducials. The machine vision application uses the fiducials to align the board for chip placement from a chip mounting device.

Gauging applications first locate and then measure, or gauge, the dimensions of an object in an image. If the measurement falls within a tolerance range, the object passes inspection. If it falls outside the tolerance range, the object is rejected. Searching for and finding image features is the key processing task that determines the success of many gauging applications such as inspecting the leads on a quad pack or inspecting an antilock-brake sensor. In real-time applications, search speed is critical.

#### 12.4.3 Geometric Matching

Geometric matching locates regions in a grayscale image that match a model or template of a reference pattern. Geometric matching is specialized to locate templates that are characterized by distinct geometric or shape information. When using geometric matching, you create a template that represents the object for which you are searching. Your machine vision application then searches for instances of the template in each inspection image and calculates a score for each match. The score relates how closely the template resembles the located matches. Geometric matching finds template matches regardless of lighting variation, blur, noise, occlusion and geometric transformations such as shifting, rotation or scaling of the template.

Geometric matching helps you quickly locate objects with good geometric information in an inspection image. You can use geometric matching in the following application areas:

- **Gauging**—Measures lengths, diameters, angles and other critical dimensions. If the measurements fall outside set tolerance levels, the object is rejected. Use geometric matching to locate the object or areas of the object you want to gauge. Use information about the size of the object to preclude geometric matching from locating objects whose sizes are too big or small.
- **Inspection**—Detects simple flaws such as scratches on objects, missing objects, or unreadable print on objects. Use the occlusion score returned by geometric matching to determine if an area of the object under inspection is missing. Use the curve matching scores returned by geometric matching to compare the boundary (or edges) of a reference object to the object under inspection.
- **Alignment**—Determines the position and orientation of a known object by locating points of reference on the object or characteristic features of the object.
- **Sorting**—Sorts objects based on shape and/or size. Geometric matching returns the location, orientation and size of each object. You can use the location of the object to pick up the object and place it into the correct bin. Use geometric matching to locate different types of objects even when objects may partially occlude each other.

Because geometric matching is an important tool for machine vision applications, it must work reliably under various, sometimes harsh, conditions. In automated machine vision applications—especially those incorporated into the manufacturing process—the visual appearance of materials or components under inspection can change because of factors such as varying part orientation, scale and lighting. The geometric matching tool must maintain its ability to locate the template patterns despite these changes. Searching and matching algorithms such as pattern matching or geometric matching, find regions in the inspection image that contain information similar to the information in the template. This information, after being synthesized, becomes the set of features that describes the image. Pattern matching and geometric matching algorithms use these sets of features to find matches in inspection images.

#### 12.4.4 Dimensional Measurements

You can use dimensional measurements or *gauging* tools in IMAQ Vision to obtain quantifiable, critical distance measurements—such as distances, angles, areas, line fits, circular fits and quantities. These measurements can help you to determine if a product was manufactured correctly. Components such as connectors, switches and relays are small and manufactured in high quantity. Human inspection of these components is tedious, time consuming and inconsistent. IMAQ Vision can quickly and consistently measure certain features on these components and generate a report of the results. If the gauged distance or count does not fall within user-specified tolerance limits, the component or part fails to meet production specifications and should be rejected.

Use gauging for applications in which inspection decisions are made on critical dimensional information obtained from image of the part. Gauging is often used in both inline and offline production. During inline processes, each component is inspected as it is manufactured. Inline

gauging inspection is often used in mechanical assembly verification, electronic packaging inspection, container inspection, glass vial inspection and electronic connector inspection. You also can use gauging to measure the quality of products off-line. First, a sample of products is extracted from the production line. Then, using measured distances between features on the object, IMAQ Vision determines if the sample falls within a tolerance range. Gauging techniques also allow you to measure the distance between particles and edges in binary images and easily quantify image measurements.

The gauging process consists of the following four steps:

- Step 1:** Locate the component or part in the image.
- Step 2:** Locate features in different areas of the part.
- Step 3:** Make measurements using these features.
- Step 4:** Compare the measurements to specifications to determine if the part passes inspection.

#### **12.4.5 Color Inspection**

Color inspection provides information about color spaces, the color spectrum, color matching, color location and color pattern matching. *Color spaces* allow you to represent a color. A color space is a subspace within a 3D coordinate system where each color is represented by a point. You can use color spaces to facilitate the description of colors between persons, machines or software programs. Various industries and applications use a number of different color spaces. Humans perceive color according to parameters such as brightness, hue and intensity, while computers perceive color as a combination of red, green and blue. The printing industry uses cyan, magenta and yellow to specify color. The following is a list of common color spaces.

- RGB—Based on red, green and blue. Used by computers to display images.
- HSL—Based on hue, saturation and luminance. Used in image processing applications.
- CIE—Based on brightness, hue and colorfulness. Defined by the Commission Internationale de l'Eclairage (International Commission on Illumination) as the different sensations of color that the human brain perceives.
- CMY—Based on cyan, magenta, and yellow. Used by the printing industry.
- YIQ—Separates the luminance information (Y) from the color information (I and Q). Used for TV broadcasting.

You must define a color space every time you process color images. With IMAQ Vision, you specify the color space associated with an image when you create the image. IMAQ Vision supports the RGB and HSL color spaces.

If you expect the lighting conditions to vary considerably during your color machine vision application, use the HSL color space. The HSL color space provides more accurate color information than the RGB space when running color processing functions such as color matching, color location and color pattern matching. IMAQ Vision's advanced algorithms for color processing—which perform under various lighting and noise conditions—process images in the HSL color space. If you do not expect the lighting conditions to vary considerably during your application, and you can easily define the colors you are looking for using red, green and blue, use the RGB space. Also

use the RGB space if you want only to display color images, but not process them in your application. The RGB space reproduces an image as you would expect to see it. IMAQ Vision always display color images in the RGB space. If you create an image in the HSL space, IMAQ Vision automatically converts the image to the RGB space before displaying it.

Because color is the brain's reaction to a specific visual stimulus, color is best described by the different sensations of color that the human brain perceives. The color-sensitive cells in the eye's retina sample color using three bands that correspond to red, green and blue light. The signals from these cells travel to the brain where they combine to produce different sensations of colors. The Commission Internationale de l'Eclairage has defined the following sensations:

- **Brightness**—The sensation of an area exhibiting more or less light
- **Hue**—The sensation of an area appearing similar to a combination of red, green and blue
- **Colorfulness**—The sensation of an area appearing to exhibit more or less of its hue
- **Lightness**—The sensation of an area's brightness relative to a reference white in the scene
- **Chroma**—The colorfulness of an area with respect to a reference white in the scene
- **Saturation**—The colorfulness of an area relative to its brightness

The trichromatic theory describes how three separate lights—red, green and blue—can be combined to match any visible color. This theory is based on the three color sensors that the eyes use. Printing and photography use the trichromatic theory as the basis for combining three different colored dyes to reproduce colors in a scene. Similarly, computer color spaces use three parameters to define a color. Most color spaces are geared toward displaying images with hardware, such as color monitors and printers, or toward applications that manipulate color information such as computer graphics and image processing. Color CRT monitors, the majority of color-video cameras and most computer graphics systems use the RGB color space. The HSL space, combined with RGB and YIQ, is frequently used in applications that manipulate color such as image processing. The color picture publishing industry uses the CMY color space also known as CMYK. The YIQ space is the standard for color TV broadcast.

*Color matching* quantifies which colors and how much of each color exist in a region of an image and uses this information to check if another image contains the same colors in the same ratio. Color matching is performed in two steps. In the first step, the machine vision software learns a reference color distribution. In the second step, the software compares color information from other images to the reference image and returns a score as an indicator of similarity.

Use *color location* to quickly locate known color regions in an image. With color location, you create a model or template that represents the colors that you are searching. Your machine vision application then searches for the model in each acquired image and calculates a score for each match. The score indicates how closely the color information in the model matches the color information in the found regions. Color can simplify a monochrome visual inspection problem by improving contrast or separating the object from the background. Color location algorithms provide a quick way to locate regions in an image with specific colors. Use color location when your application has the following characteristics:

- Requires the location and the number of regions in an image with their specific color information
- Relies on the cumulative color information in the region, instead of how the colors are arranged in the region

- Does not require the orientation of the region
- Does not require the location with subpixel accuracy

The color location tools in IMAQ Vision measure the similarity between an idealized representation of a feature, called a model, and a feature that may be present in an image. A feature for color location is defined as a region in an image with specific colors. Color location is useful in many applications. Color location provides your application with information about the number of instances and locations of the template within an image. Use color location in the following general applications—inspection, identification and sorting.

*Inspection* detects flaws such as missing components, incorrect printing and incorrect fibers on textiles. A common pharmaceutical inspection application is inspecting a blister pack for the correct pills. Blister pack inspection involves checking that all the pills are of the correct type, which is easily performed by checking that all the pills have the same color information. Because your task is to determine if there are a fixed number of the correct pills in the pack, color location is a very effective tool.

*Identification* assigns a label to an object based on its features. In many applications, the color-coded identification marks are placed on the objects. In these applications, color matching locates the color code and identifies the object. In a spring identification application, different types of springs are identified by a collection of color marks painted on the coil. If you know the different types of color patches that are used to mark the springs, color location can find which color marks appear in the image. You then can use this information to identify the type of spring.

*Sorting* separates objects based on attributes such as color, size and shape. In many applications, especially in the pharmaceutical and plastic industries, objects are sorted according to color, such as pills and plastic pellets. Color location is built upon the color matching functions to quickly locate regions with specific color information in an image.

#### 12.4.6 Binary Partical Classification

Binary particle classification identifies an unknown binary *sample* by comparing a set of its significant *features* to a set of features that conceptually represent *classes* of known samples. *Classification* involves two phases: training and classifying.

*Training* is a phase during which you teach the machine vision software the types of samples you want to classify during the classifying phase. You can train any number of samples to create a set of classes which you later compare to unknown samples during the classifying phase. You store the classes in a *classifier* file. Training might be a one-time process, or it might be an incremental process you repeat to add new samples to existing classes, or to create several classes and thus broadening the scope of samples you want to classify.

*Classifying* is a phase during which your custom machine vision application classifies an unknown sample in an inspection image into one of the classes you trained. The classifying phase classifies a sample according to how similar the sample features are to the same features of the trained samples.

The need to classify is common in many machine vision applications. Typical applications involving particle classification include the following:

- **Sorting**—Sorts samples of varied shapes. For example, a *particle classifier* can sort different mechanical parts on a conveyor belt into different bins. Example outputs of a sorting or identification application could be user-defined labels of certain classes.

- **Inspection**—Inspects samples by assigning each sample an identification score and then rejecting samples that do not closely match members of the training set. Example outputs of a sample inspection application could be *Pass* or *Fail*.

#### 12.4.7 Optical Character Recognition

Optical Character Recognition (OCR) provides machine vision functions which you can use in an application to perform OCR. OCR is the process by which the machine vision software reads text and/or characters in an image. NI OCR consists of the following two parts:

- An application for training *characters*
- Tools such as the NI Vision Builder for Automated Inspection software or libraries of LabVIEW VIs, LabWindows/CVI functions, and Microsoft Visual Basic properties and methods.

Use these tools to create a machine vision application that analyzes an image and compares objects in that image to the characters you trained to determine if they match. The machine vision application returns the matching characters that it read.

Training characters is the process by which you teach the machine vision software the types of characters and/or patterns you want to read in the image during the reading procedure. You can use OCR to train any number of characters creating a character set. The set of characters is later compared with objects during the reading procedure. You store the character set in a character set file. Training might be a one-time process, or it might be a process you repeat several times creating several character sets to broaden the scope of characters which you want to detect in an image. Reading characters is the process by which the machine vision application you create analyzes an image to determine if the objects match the characters you trained. The machine vision application reads characters in an image using the character set that you created when you trained characters.

Typically, machine vision OCR is used in automated inspection applications to identify or classify components. For example, you can use OCR to detect and analyze the serial number on an automobile engine that is moving along a production line. Using OCR in this instance helps you identify the part quickly, which in turn helps you quickly select the appropriate inspection process for the part. You can use OCR in a wide variety of other machine vision applications such as the following:

- Inspecting pill bottle labels and lot codes in pharmaceutical applications
- Verifying wafers and IC package codes in semiconductor applications
- Controlling the quality of stamped machine parts
- Sorting and tracking mail packages and parcels
- Reading alphanumeric characters on automotive parts

#### 12.4.8 Instrument Readers

Instrument readers are functions which you can use to accelerate the development of applications that require reading meters, seven segment displays and barcodes. Use instrument readers when you need to obtain information from images of simple meters, LCD displays and barcodes.

Meter functions simplify and accelerate the development of applications that require reading values from meters or gauges. These functions provide high-level vision processes to extract the position of a meter or gauge needle. You can use this information to build different applications such as the calibration of a gauge. Use the functions to compute the base of the needle and its extremities from an area of interest indicating the initial and the full-scale position of the needle. You then can use these VIs to read the position of the needle using parameters computed earlier. The recognition process consists of the following two phases:

- A *learning phase* during which the user must specify the extremities of the needle.
- An *analysis phase* during which the current position of the needle is determined. The meter functions are designed to work with meters or gauges that have either a dark needle on a light background or a light needle on a dark background.

LCD functions simplify and accelerate the development of applications that require reading values from seven-segment displays. Use these functions to extract seven-segment digit information from an image.

The reading process consists of two phases.

- A *learning phase* during which the user specifies an area of interest in the image to locate the seven-segment display.
- A *reading phase* during which the area specified by the user is analyzed to read the seven-segment digit.

The IMAQ Vision LCD functions provide the high-level vision processes required for recognizing and reading seven-segment digit indicators. The VIs in this library are designed for seven-segment displays that use either LCDs or LEDs composed of electroluminescent indicators or light-emitting diodes, respectively. The functions in this library can perform the following tasks:

- Detect the area around each seven-segment digit from a rectangular area that contains multiple digits.
- Read the value of a single digit.
- Read the value, sign and decimal separator of the displayed number.

## 12.5 MACHINE VISION HARDWARE AND SOFTWARE

National Instruments hardware consists of Image acquisition card, analog or digital camera and lens. National Instruments software combines LabVIEW 8.0 or later and NI IMAQ driver software. The Vision Application Software consists of Vision Builder for automated inspection to configure, benchmark and deploy without programming. The Vision Development Module includes NI Vision Assistant and hundreds of machine vision and image processing functions for complete control and functionality in an application development environment such as LabVIEW, C/C++, Visual Basic.

National Instruments IMAQ devices feature image acquisition boards that connect to parallel digital, analog and camera link cameras. These devices include advanced triggering and digital I/O features that you can use to trigger an acquisition from an outside signal. IMAQ devices feature

up to 128 MB of onboard memory which allows you to acquire images at extremely high rates while sustaining high-speed throughput and greater overall system performance. Most IMAQ devices work with motion control and data acquisition hardware using the National Instruments real-time system integration bus (RTSI). On National Instruments PCI boards, you can use a ribbon cable to connect RTSI connectors on adjacent boards to send triggering and timing information from one board to another. Real-Time Acquisition is available with LV RT on a PXI chassis or with LV RT and Vision Builder for AI on a CVS (1450 Series).

NI-IMAQ driver software features an extensive library of functions that you can call from your application programming environment. These functions include routines for video configuration, image acquisition (continuous or single-shot), memory buffer allocation, trigger control and board configuration. NI-IMAQ also provides all the functionality you need to acquire and save images.

## 12.6 BUILDING A COMPLETE MACHINE VISION SYSTEM

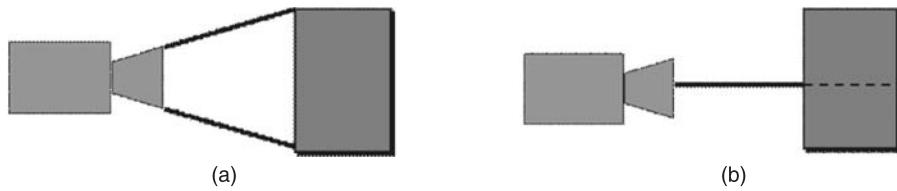
IMAQ Vision functions allow you to accelerate the development of industrial machine vision and scientific imaging systems using routines for grayscale, binary and color image display; image processing, including statistics, filtering and geometric transforms; pattern, shape and color matching; blob analysis; gauging and measurement.

Use IMAQ Vision Assistant to prototype and test your IMAQ Vision applications. IMAQ Vision Assistant uses the IMAQ Vision function library, but performs as a stand alone executable tool that you can use independently of other programs. You can use IMAQ Vision Assistant to learn how vision functions work together and save the steps of your prototype in a script from which you can re-create the application in another programming environment.

Use Measurement & Automation Explorer (MAX) to configure your IMAQ device and acquire your first images. Use MAX to set up your camera and hardware, establish acquisition parameters, configure your acquisition, and even execute and save an acquisition. The functions of MAX are select your camera, set parameters for regions of interest, filters, asynchronous acquisition, and exposure time and set up acquisitions from non-interlaced progressive scan cameras.

Before you acquire images, you must set up your imaging system. If objects in your image are covered by shadows or glare, it becomes much more difficult to examine the images effectively. Some objects reflect large amounts of light due to the nature of their external coating or their curvature. Poor lighting setups in the imaging environment can create shadows that fall across the image. When possible, position your lighting setup and your imaged object such that glare and shadows are reduced or eliminated. If this is not possible, you may need to use special lighting filters or lenses.

The two types of cameras used are area scan and line scan [Figures 12.2(a) and (b)]. The *area scan* scans one area of pixels at a time. It slows acquisition but acquires the entire image. The advantages are less processing, less expensive and so used in most applications. The *line scan* scans one line of pixels at a time. It is used for faster acquisition and you need to put the lines together in software to make a picture. The features are more processing, more expensive and good for objects on an assembly line or rotating cylindrical objects.



**Figure 12.2** (a) Area scan and (b) Line scan.

Cameras can be classified as analog and digital. Analog cameras output video signals in an analog format. For most low-end cameras, the odd and even fields are interlaced to increase the perceived image update rate, a technique that has been used by the television industry for several years. Two fields are combined to make up a frame. Digital camera provides high image quality and pixel depth. The digitizer is housed inside the camera. It has large image sizes and frame rates. Digital cameras use three types of signals—data lines, a pixel clock and enable lines. Data lines are parallel wires that carry digital signals corresponding to pixel values. Digital cameras typically represent pixels with 8, 10, 12, or 14 bits. Color digital cameras can represent pixels with up to 24 bits. Depending on your camera, you may have as many as 24 data lines representing each pixel. The pixel clock has a high-frequency pulse train that determines when the data lines contain valid data. On the active edge of the pixel clock, digital lines have a constant value that is input to your IMAQ device. The pixel clock frequency determines the rate that pixels are acquired. Enable lines indicate when data lines contain valid data. When choosing between digital and analog cameras, keep in mind that digital cameras tend to be more expensive than analog cameras, but they allow faster frame rates, higher bit and spatial resolution, and higher signal-to-noise ratios. Analog cameras are based on older, proven technology and are therefore more common and less expensive, and their standardization allows for simpler cabling and easier setup. A low-cost, monochrome analog camera is an appropriate choice for most beginner vision applications.

## 12.7 ACQUIRING AND DISPLAYING IMAGES WITH NI-IMAQ DRIVER SOFTWARE

NI-IMAQ is a complete and robust API for image acquisition. Whether you are using LabVIEW, Measurement Studio, Visual Basic, or Visual C++, NI-IMAQ gives you high-level control of National Instruments image acquisition devices. NI-IMAQ performs all of the computer- and board-specific tasks, allowing straightforward image acquisition without register-level programming. NI-IMAQ is compatible with NI-DAQ and all other National Instruments driver software for easily integrating an imaging application into any National Instruments solution. NI-IMAQ is included with your hardware at no charge.

NI-IMAQ features an extensive library of functions as in Figure 12.3 that you can call from your application programming environment. These functions include routines for video configuration, image acquisition (continuous and single-shot), memory buffer allocation, trigger control and board configuration. NI-IMAQ performs all functionality required to acquire and save images. For image analysis functionality, refer to the IMAQ Vision software analysis libraries which are discussed later in this course. NI-IMAQ resolves many of the complex issues between the computer and IMAQ hardware internally, such as programming interrupts and DMA controllers.

NI-IMAQ also provides the interface path between LabVIEW, Measurement Studio, or other programming environments and the hardware product.



Figure 12.3 NI-IMAQ functions.

NI-IMAQ and IMAQ Vision use five categories of functions to acquire and display images:

- **Utility functions**—Allocate and free memory used for storing images; begin and end image acquisition sessions
- **Single buffer acquisition functions**—Acquire images into a single buffer using the snap and grab functions
- **Multiple buffer acquisition functions**—Acquire continuous images into multiple buffers using the ring and sequence functions
- **Display controls**—Display images for processing
- **Trigger functions**—Link a vision function to an event external to the computer, such as receiving a pulse to indicate the position of an item on an assembly line

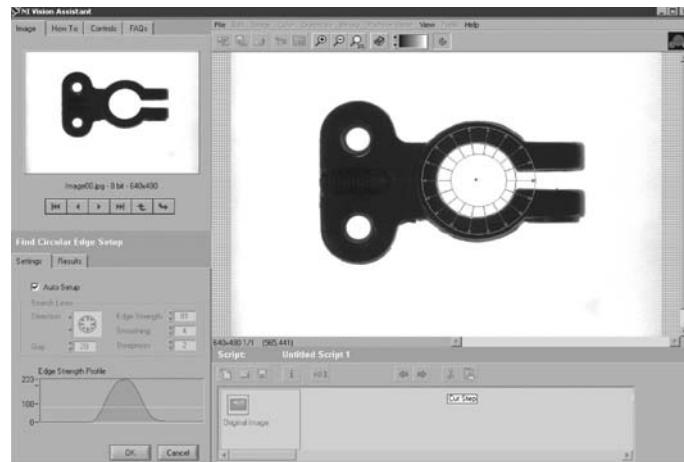
Snaps and grabs are the most basic types of acquisitions. A snap is simply a snapshot, in which you acquire a single image from the camera. A grab is more like a video, in which you acquire every image that comes from the camera. The images in a grab are displayed successively, producing a full-motion video, consisting of around 25 to 30 frames per second. IMAQ Snap, IMAQ Grab Setup and IMAQ Grab Acquire are used to snap and grab images.

## 12.8 IMAGE PROCESSING TOOLS AND FUNCTIONS IN IMAQ VISION

Utility functions include VIs for image management and manipulation, file management, calibration and region of interest processing. Image processing functions include VIs for analysis, color processing, frequency processing, filtering, morphology, operations, and processing, including

IMAQ Histogram, IMAQ Threshold and IMAQ Morphology. Machine vision VIs are used for common inspection tasks such as checking for the presence or absence of parts in an image or measuring dimensions in comparison to specifications. Some examples of the machine vision VIs are the caliper and coordinate system VIs.

Image processing is a time-consuming process, both in computer processor time and development time. National Instruments has developed an application to accelerate the design time of a machine vision application—IMAQ Vision Assistant as shown in Figure 12.4. IMAQ Vision Assistant allows even the first-time vision developer to learn image processing techniques and test inspection strategies. In addition, more experienced developers can develop and explore vision algorithms faster with less programming.



**Figure 12.4** IMAQ Vision Assistant.

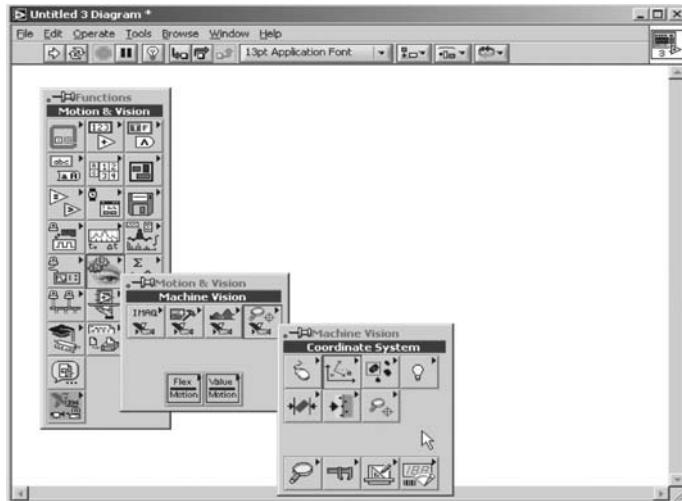
The Machine Vision VIs are conveniently located in the Machine Vision Palette as shown in Figure 12.5. They are broken down into separate sub-palettes: Select Region of Interest, Coordinate System, Count and Measure Objects, Measure Intensities, Measure Distances, Locate Edges, Find Patterns, Searching and Matching, Caliper, Analytic Geometry and Instrument Readers.

Most of the VIs you will use fall into these function categories:

- Statistics
- Particle analysis
- Pattern matching
- Edge detection
- Gauging

Statistical image processing functions as shown in Figure 12.6 gives a representation of the image, such as the average pixel value or the standard deviation of the grayscale values. In an inspection application, under strict lighting conditions, you can calculate the average intensity of a region of interest (ROI) to quickly determine if a chip is present or not. The average grayscale value will vary depending on if the chip is present. In less constrained environments, you can use the standard deviation of intensity in an ROI the same way. You can use the ROI control to draw a

line on the image; the pixels on the line make up the line profile. Line profiles are useful for making a 2D graph of the pixel intensity values or for finding distances between objects and their edges. The average, standard deviation, histogram and line profile functions are all part of the IMAQ Vision software package.



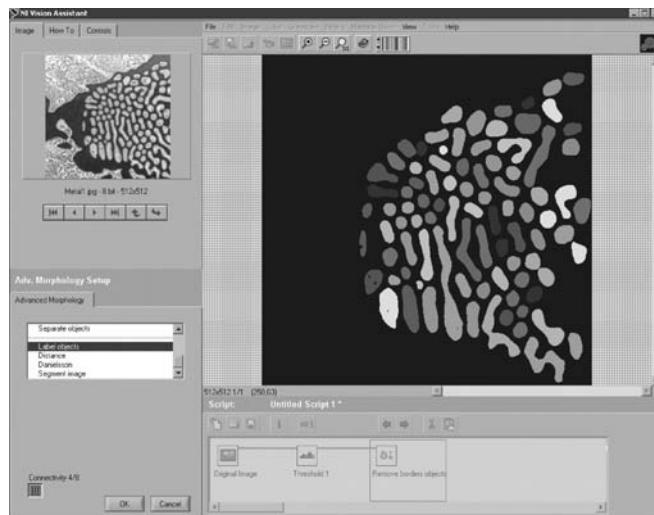
**Figure 12.5** Programming with IMAQ Vision's Machine Vision VIs.



**Figure 12.6** Statistical image processing.

You can use particle analysis functions as in Figure 12.7 which help to find dozens of parameters for a binary large object, also known as a *blob*, including the orientation, area, perimeter, center of mass and coordinates of each blob. You can use these unique parameters to identify parts in a sorting application. You can also use them to ensure that a manufactured part meets quality standards. A subset of the particle analysis functions deals with morphology. These functions are

extremely useful in counting and inspection applications. You can use these functions to change the shape of the image particles to make counting easier. For example, you can use the erosion function to erode the perimeter of objects, allowing you to accurately count two particles that are close to each other or overlapping. In addition, the morphology functions can remove small particles such as an unwanted blob caused by reflections in the image.

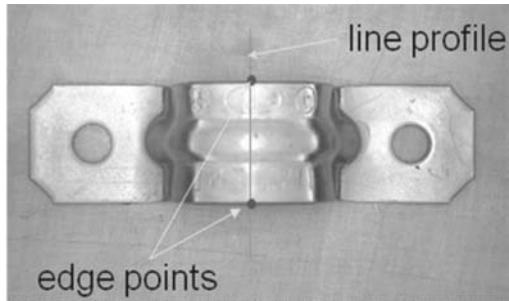


**Figure 12.7** Particle analysis function.

Pattern matching provides information about the presence or absence, number and location of a user-defined template within an image. For example, you can search an image containing a printed circuit board for one or more alignment marks called *fiducials*. You can then use the positions of the marks to align the board for chip placement by a chip-mounting device. You can also use pattern matching to locate key components in gauging applications.

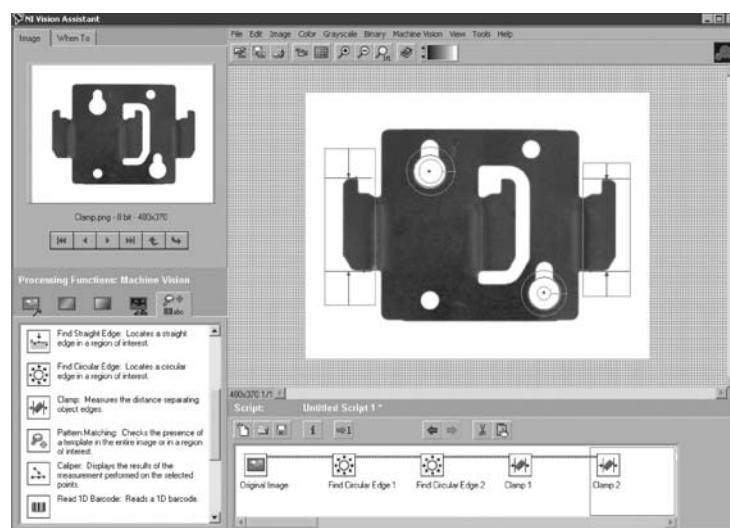
Edge detection as shown in Figure 12.8 is an important image processing and machine vision function. This function clearly determines the boundary or edge of an object and is represented by intensity discontinuities in an image. Once the software locates the edges in the image, you can use those edge locations to compute the distances between them. Since edge detection works only on one-dimensional data, it is useful for finding object boundaries or other areas of significant intensity changes. Edge detection can help identify a crack in an object, while finding parallel edges in a soldering application can help determine the quality of the solder application. The edge information in pattern recognition is very important, since you can often recognize an object from only a crude outline. IMAQ Vision includes many different edge detection functions, including as Sobel Filter, Sigma, Roberts Filter, Prewitt Filter, Gradient and Differentiation.

You can use gauging functions to automatically measure distances and angles between edges of an object. Measure from point-to-point using the line profile and interactive display window. You can also specify markers and calculate measurements between markers. In a production application, you can measure critical distances and compare them to a user-defined tolerance in order to reject flawed products. You can also find the position, angle and sharpness of an edge for calculating edge distance measurements with sub-pixel accuracy.



**Figure 12.8** Edge detection.

There are many more IMAQ vision functions as shown in Figure 12.9. IMAQ Vision is designed for ease-of-use and specifically built to meet the requirements of imaging application developers under pressure to reduce cost and time to market. Transparent memory management, logically named VIs, functions and parameters make IMAQ Vision easy to learn. The high-level functions are built to work together intuitively so you can develop faster using only a few functions. In addition to the functions covered, IMAQ Vision offers a variety of other image processing functions. You can also use IMAQ Vision image processing functions to filter, manipulate, smooth and quantify images. Arithmetic operations include add, subtract, multiply and divide. There are also local operations such as NOT, AND, OR, XOR and compare. Filtering functions include threshold, auto threshold, lowpass filter (Gaussian), median filter, edge detection and custom filters of any kernel size. Use the IMAQ Advance color analysis functions to identify, measure, compare and match colors. These color analysis functions are ideal for accurately judging color quality in a variety of applications, including automotive, pharmaceutical and printing. You can rotate, resample and equalize images by using the line profile tool to measure pixel values and find defects along any arbitrary line in an image.

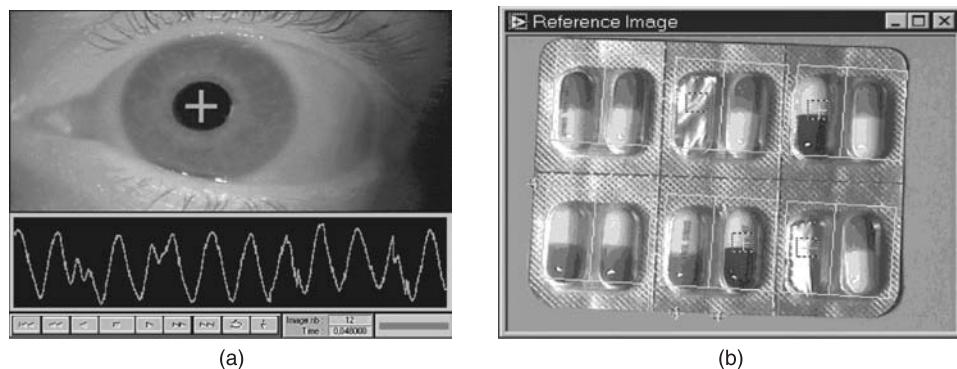


**Figure 12.9** IMAQ vision functions.

## 12.9 MACHINE VISION APPLICATION AREAS

Machine Vision application areas are Automotive, Biomedical, Electronics, Pharmaceuticals, Industrial Automation, Manufacturing Scientific.

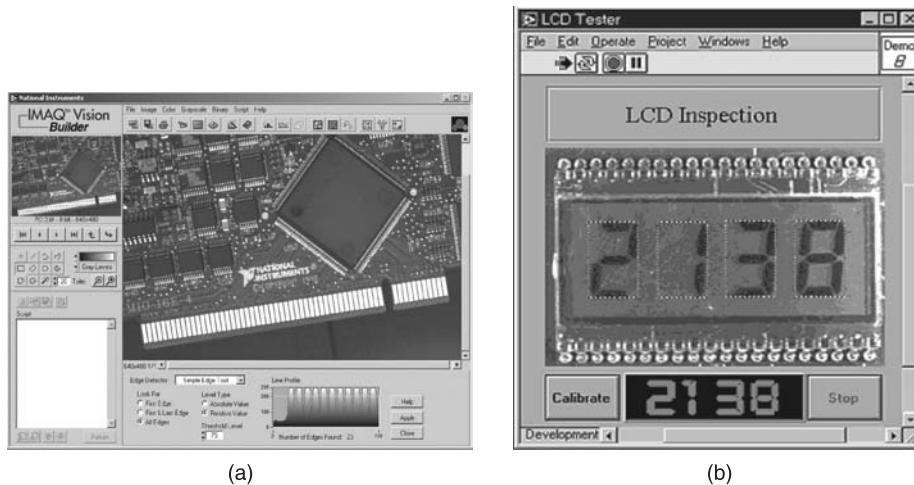
Engineers in the automotive industry use National Instruments LabVIEW software, signal conditioning, DAQ hardware and IMAQ Vision software to control, monitor, and provide sophisticated analysis functions for testing airbag deployment, inspecting fuse boxes, and calibrating meters. Medical doctors and biomedical engineers use National Instruments imaging products for various applications such as counting and measuring the velocity of cells under a microscope and tracking the pupil of the human eye in response to certain stimuli [Figure 12.10(a)]. Quality assurance and industrial engineers in the pharmaceutical industry use National Instruments machine vision tools for various applications such as blister pack inspection to ensure that each package is full and contains the correct capsules, as well as foreign tablet inspection to verify that prescription drugs are properly bottled and labeled in order to ensure patient safety [Figure 12.10(b)].



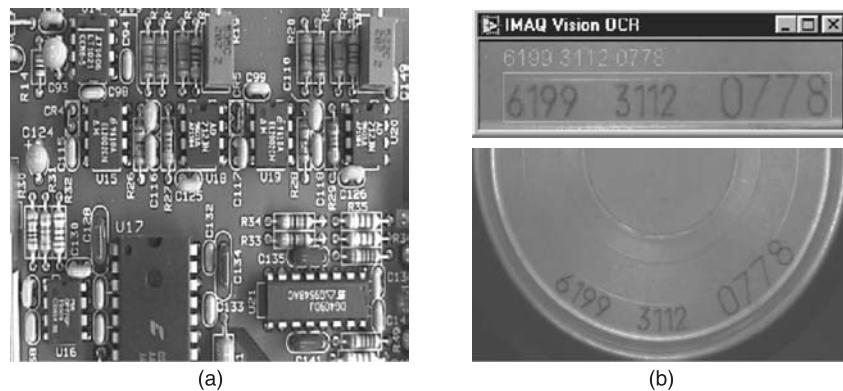
**Figure 12.10** (a) Biomedical—tracking eye movement in response to stimuli and (b) pharmaceuticals—blister pack inspection and foreign tablet inspection.

National Instruments imaging products serve many purposes in the electronics industry. Using color image processing, color matching and powerful inspection functions, developers can automate dozens of applications, from the verification of part placement on printed circuit boards to the inspection of liquid crystal displays as shown in Figures 12.11(a) and (b). Integrated barcode inspection algorithms help lower development costs, since all the software required for your machine vision application is integrated into IMAQ Vision. A top cola manufacturer discovered the benefits of computer-based machine vision in their application to inspect the containers for their soda. Other happy customers have given feedback regarding their experiences with IMAQ Vision.

Manufacturing and test engineers verify their production runs using National Instruments machine vision tools. You can test printed circuit boards for correct traces [Figure 12.12(a)], connections and part placement. You can also inspect supply canisters for proper serial number labeling [Figure 12.12(b)]. Research and development solutions that benefit from the National Instruments IMAQ product line include heat transfer analysis using an infrared camera and iron ore analysis for the purposes of verifying the purity concentration of certain elements.



**Figure 12.11** (a) Electronics—resistor placement verification and Inspection of liquid crystal displays and (b) inspection of liquid crystal display.



**Figure 12.12** Manufacturing—(a) Inspection of printed circuit boards and (b) Identification of supply canisters.

## SUMMARY

- National Instruments IMAQ devices include PCI/PXI Image Acquisition Cards (Frame Grabbers), Compact Vision System and Smart Camera.
- IMAQ software includes NI-IMAQ Driver Software, Vision Development Module, Vision Builder for Automated Inspection and Vision Assistant.
- Use Measurement & Automation Explorer (MAX) to configure your IMAQ device and acquire your first images.
- Before you acquire images, you must set up your imaging system. There are several fundamental imaging parameters including resolution, sensor resolution, field of view and working distance.

- Lighting is one of the most important aspects in setting up your imaging environment.
- The area scan camera scans one area of pixels at a time. The line scan camera scans one line of pixels at a time.
- Analog cameras output video signals in analog format, whereas digital cameras produce video outputs in digital format. Analog formats include RS-170, NTSC, CCIR, PAL. Digital formats include parallel standard, IEEE 1394 standard, camera link standard.
- A snap is a one-shot acquisition into a single buffer. A grab is a continuous acquisition into a single buffer. A sequence is a series of one-shot acquisitions into multiple buffers. A ring is a series of continuous acquisitions into multiple buffers.
- IMAQ Vision functions are divided into three categories: vision utility VIs, image processing VIs, machine vision VIs.
- Most of the IMAQ Vision functions fall into the following categories: statistics, particle analysis, pattern matching, edge detection and gauging.
- Histograms indicate the number of pixels at each grayscale level. Thresholds convert all pixel values in an image to 0 or 1, according to the value of the original pixel.
- Binary: An image is segmented into two regions during the thresholding process—particle region (pixels equal to 1) and background region (pixels equal to 0).
- Morphological functions remove unwanted information caused by the thresholding process.
- Filtering alters the structure of particles. Erosion decreases the size of objects in an image. Dilation increases the size of objects in an image.
- Calibration translates pixels to real-world units, so you can make accurate measurements.
- Commonly used machine vision techniques are edge detection, regions of interest, pattern matching. Edge detection finds locations of significant intensity changes within an image. Region of Interest is an area of an image in which you want to focus your image analysis. Pattern matching locates regions of a grayscale image that match a predefined template.
- Commonly used color tools are color matching, color location and color pattern matching.
- Coordinate systems allow you to define search areas that can shift and rotate with the object you are inspecting.

## MISCELLANEOUS SOLVED PROBLEMS

**Problem 12.1** Create a VI to acquire a single image into memory buffer using IMAQ snap and save the image in a file.

**Solution** The block diagram to acquire a single image into memory buffer using IMAQ snap is shown in Figure P12.1.

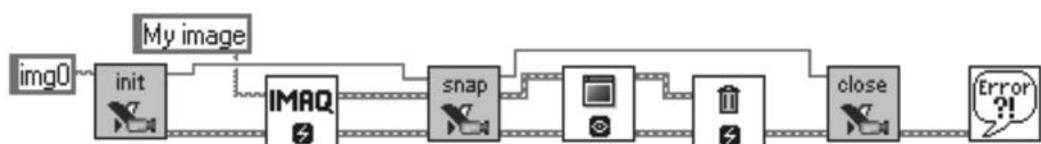


Figure P12.1 Block diagram.

**Problem 12.2** Create a VI to acquire images using a While Loop around IMAQ snap.

**Solution** The block diagram to acquire images using a While Loop around IMAQ snap is shown in Figure P12.2.

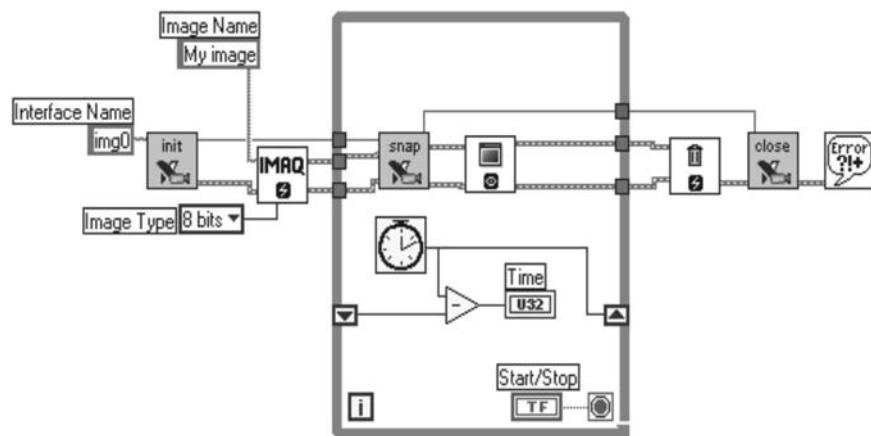


Figure P12.2 Block diagram.

**Problem 12.3** Create a VI to acquire live images using a grab and compare the acquisition rates of grab and snap.

**Solution** The block diagram to solve the problem is given in Figure P12.3.

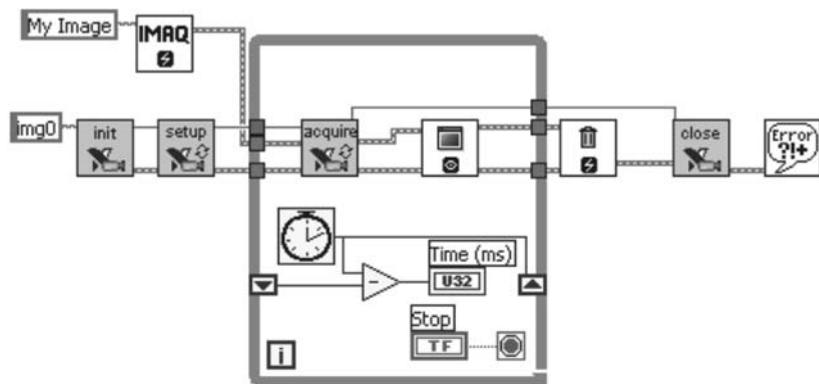
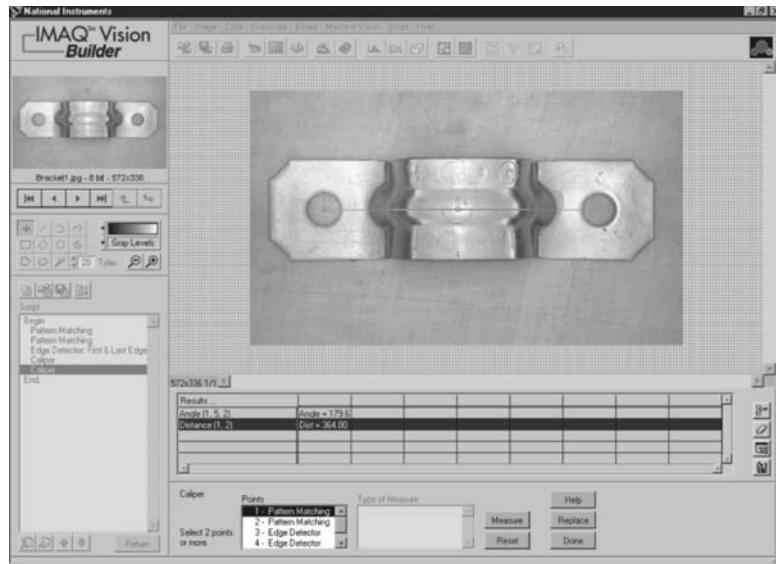


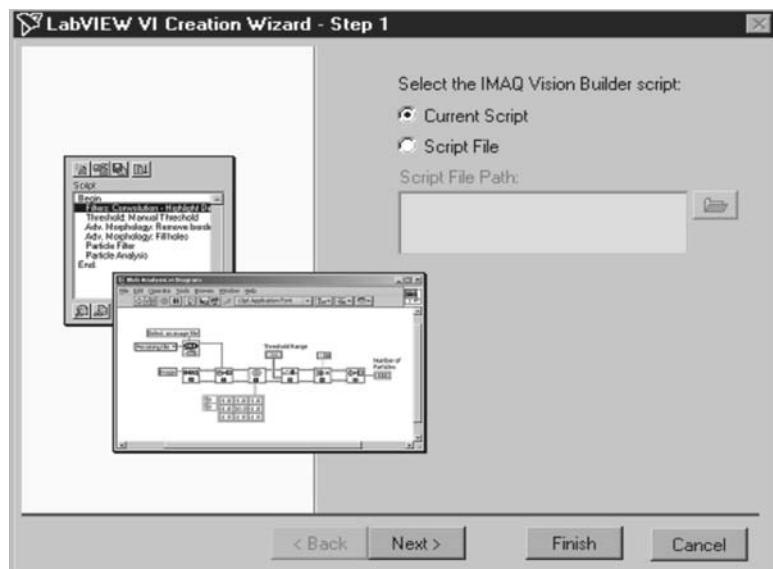
Figure P12.3 Block diagram.

**Problem 12.4** Using IMAQ Vision Builder, determine the angle of bend in a bracket. Use pattern matching, edge detection and caliper techniques. Create the LabVIEW file using the script developed using Vision Builder.

**Solution** The front panel for measuring the angle of tilt is shown in Figure P12.4(a). The LabVIEW file creation wizard is given in Figure P12.4(b).



(a) Front panel for measuring the angle of tilt

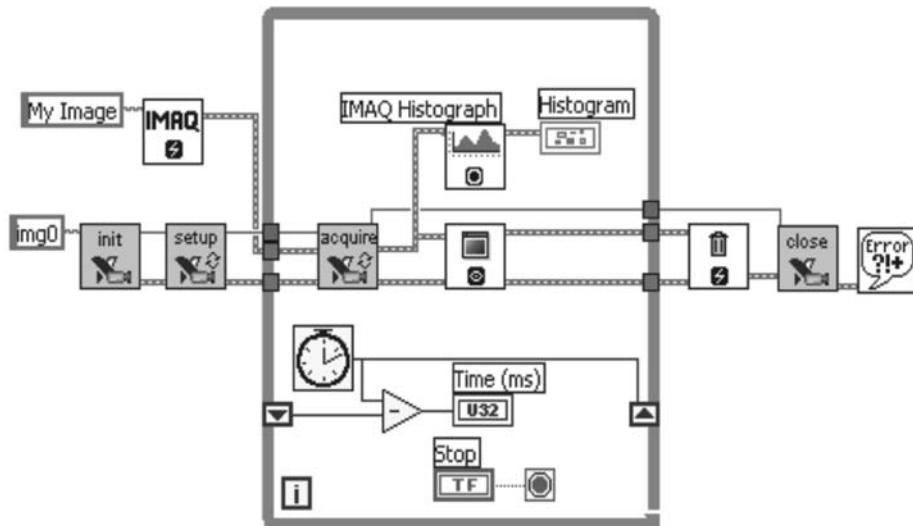


(b) LabVIEW file creation wizard

**Figure P12.4**

**Problem 12.5** Create a VI to find the histogram from the acquired image.

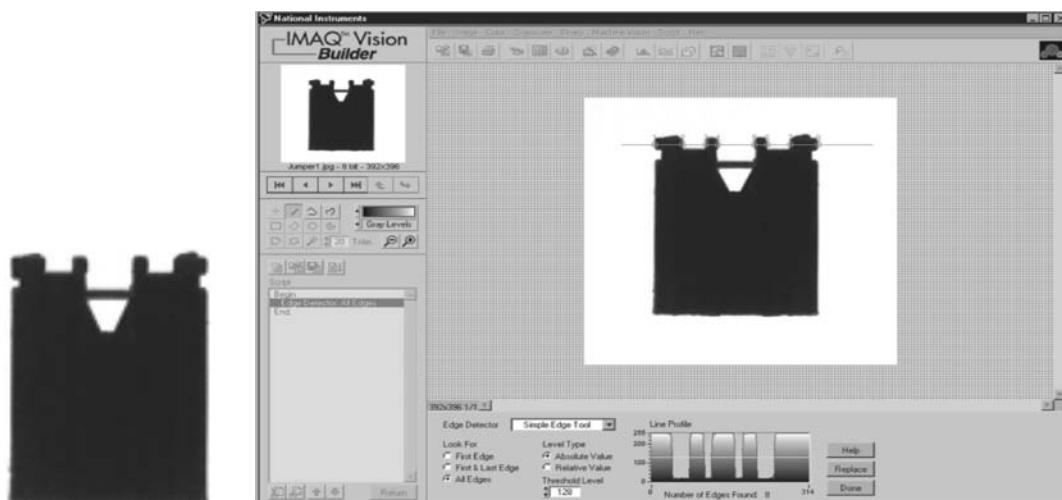
**Solution** The block diagram to create a VI and find the histogram from the acquired image is shown in Figure P12.5.



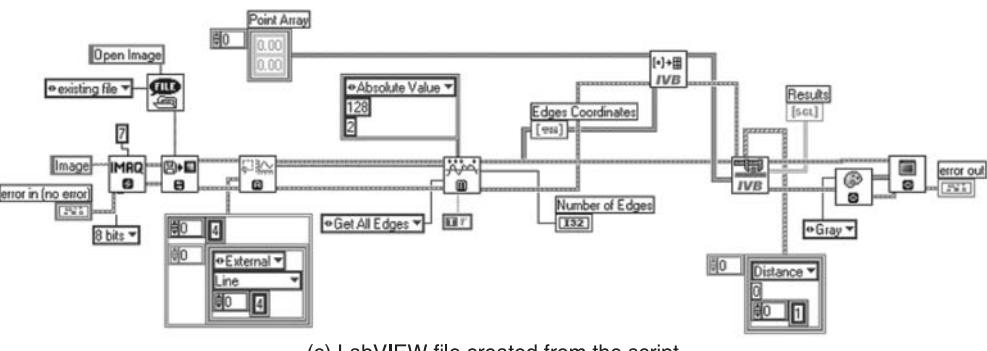
**Figure P12.5** Block diagram to find the histogram.

**Problem 12.6** Using IMAQ Vision Builder, find the distance between edges in the jumper shown below. Use edge detection and caliper tools. Create the LabVIEW file using the script developed using Vision Builder.

**Solution** Figure P12.6(a) shows the original image of the jumper and Figure P12.6(b) provides edge detection in the jumper. Figure P12.6(c) shows the block diagram of the LabVIEW file created from the script.



**Figure P12.6 (Contd.)**



(c) LabVIEW file created from the script

**Figure P12.6**


---

## REVIEW QUESTIONS

---

1. Define digital image and image resolution.
2. Explain the three types of images.
3. What is the composition and format of an image file?
4. Draw and explain the fundamental parameters of an imaging system.
5. Explain the importance of image processing and analysis in IMAQ.
6. Explain how particle measurements is about characterizing digital particles.
7. Explain the complete IMAQ product family that consists of IMAQ Hardware NI-IMAQ and IMAQ Vision.
8. Explain machine vision conceptual information about high-level operations commonly used in machine vision applications.
9. What are the NI-IMAQ and IMAQ Vision functions used to acquire and display images?
10. What are the image processing tools and functions in IMAQ vision used in developing an application?
11. Explain the application of machine vision used in industry.

---

## EXERCISES

---

1. Create a VI to acquire a single image into memory buffer using IMAQ Snap.
2. Create a VI to acquire multiple images into memory buffer using IMAQ Grab.
3. Create a VI to acquire multiple images into multiple memory buffers.
4. Acquire a selected portion of the image using the IMAQ WindDraw function.
5. Use the trigger input on an IMAQ device to acquire one image into the buffer.
6. Use the trigger input on an IMAQ device to acquire one image into the buffer. That image will display each time you press the trigger button.

7. Create a VI that can snap an image and measure its edges.
8. Using IMAQ Vision Builder, find the holes in the bracket shown in Figure E8. Use the threshold and morphology techniques. Create the LabVIEW file using the script developed using Vision Builder.

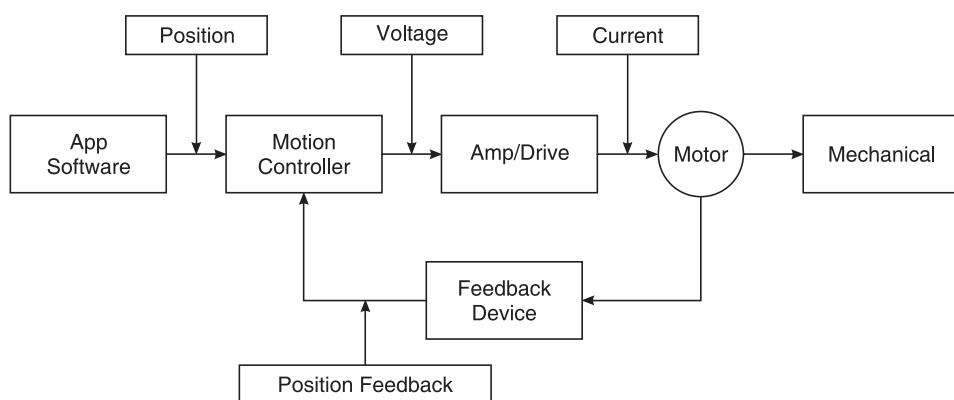


**Figure E8** Original image—bracket.

# MOTION CONTROL

## 13.1 COMPONENTS OF A MOTION CONTROL SYSTEM

Figure 13.1 shows the different components of a motion control system.



**Figure 13.1** Components of a motion control system.

**Application software:** You can use application software to command target positions and motion control profiles.

**Motion controller:** The motion controller acts as brain of the system by taking the desired target positions and motion profiles and creating the trajectories for the motors to follow, but outputting a  $\pm 10$  V signal for servomotors, or a step and direction pulses for stepper motors.

**Amplifier or drive:** Amplifiers (also called drives) take the commands from the controller and generate the current required to drive or turn the motor.

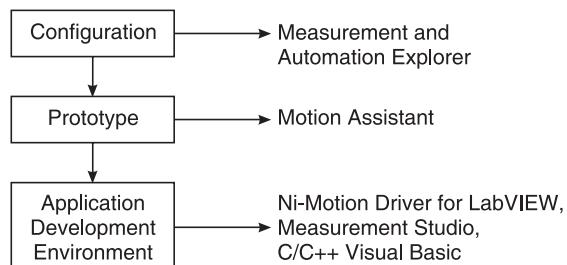
**Motor:** Motors turn electrical energy into mechanical energy and produce the torque required to move to the desired target position.

**Mechanical elements:** Motors are designed to provide torque to some mechanics. These include linear slides, robotic arms and special actuators.

**Feedback device or position sensor:** A position feedback device is not required for some motion control applications (such as controlling stepper motors), but is vital for servomotors. The feedback device, usually a quadrature encoder, senses the motor position and reports the result to the controller, thereby closing the loop to the motion controller.

## 13.2 SOFTWARE FOR CONFIGURATION, PROTOTYPING AND DEVELOPMENT

Application software is divided into three main categories—configuration, prototype and application development environment (ADE). Figure 13.2 illustrates the motion control system programming process and the corresponding National Instruments product designed for the process.



**Figure 13.2** The motion control system development process.

### 13.2.1 Configuration

One of the first things to do is configure your system. National Instruments offers Measurement and Automation Explorer, an interactive tool for configuring not only motion control, but all other National Instruments hardware. For motion control, Measurement and Automation Explorer offers interactive testing and tuning panels that help you verify your system functionality before you program. NI Measurement and Automation Explorer (MAX) shown in Figure 13.3 is an interactive tool for configuring and tuning your motion control system.

For installing and configuring FlexMotion NI 73xx controllers, you need to understand servo tune information about servo tune fundamentals, automatically tuning motors, manually tuning motors and advanced tuning techniques if necessary.

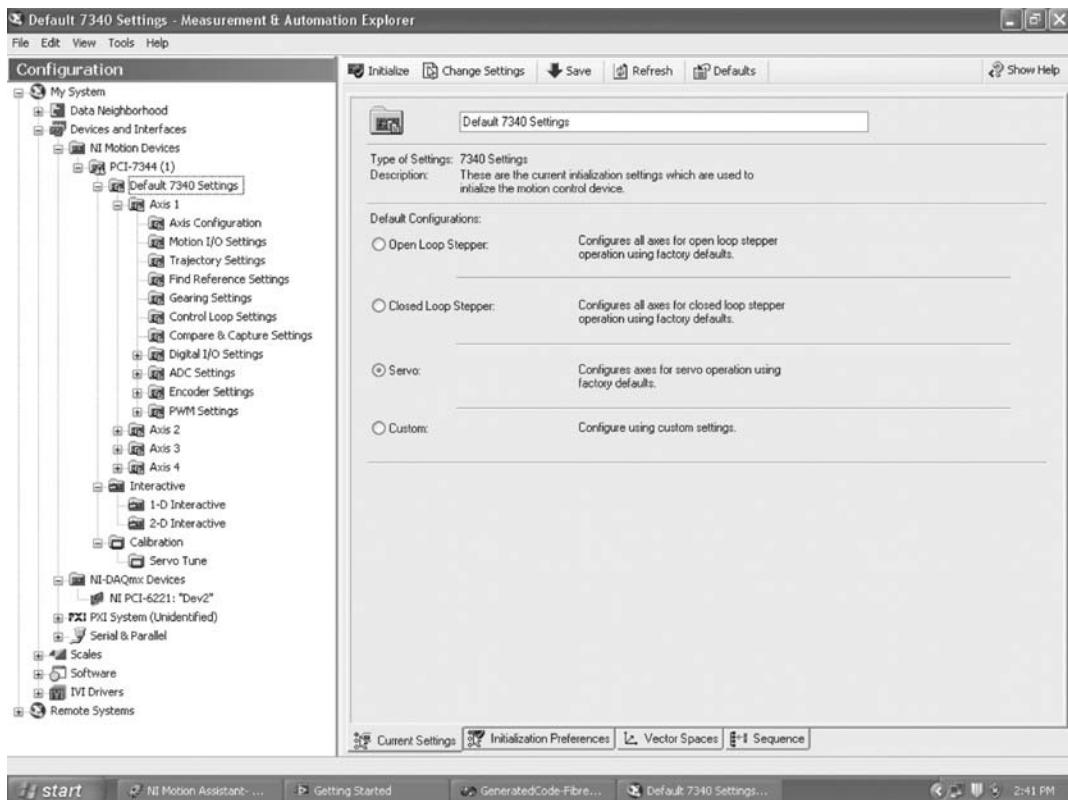


Figure 13.3 NI Measurement and Automation Explorer (MAX).

The Servo Tune document provides information necessary for getting started with tuning servo tune motors. It requires basic knowledge of servomotors, Measurement & Automation Explorer (MAX) and motion control concepts. It is divided into the following sections: Getting Started: Accessing Servo Tune, PID Control Loop Parameters, Step Response, Stability in the Time Domain and Analyzing the Step Response Plot. To get started and accessing Servo Tune, follow the steps:

**Step 1:** Launch MAX.

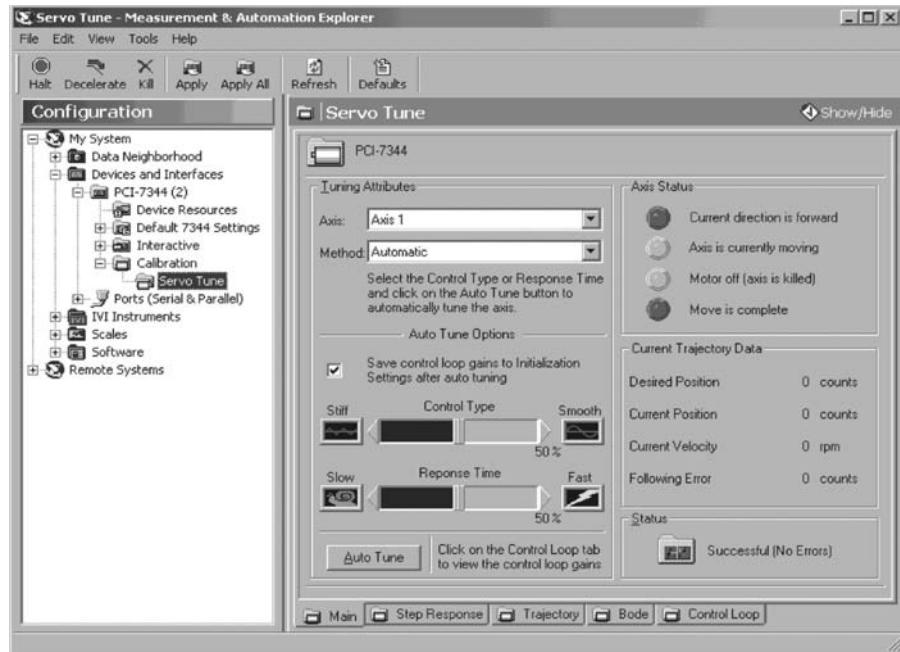
**Step 2:** Expand the *Devices and Interfaces* branch on the configuration tree.

**Step 3:** Expand *PCI-7344*.

**Step 4:** Expand *Calibration*, and click *Servo Tune*.

Figure 13.4 shows the Servo Tune interface.

When automatically tuning your system, use the **Control Type** and **Response Time** parameters to customize your system. Smooth controls have a slower response time. The smoother the control, the less the axis will overshoot its desired position before slowing. The more overshoot the system can manage, the faster the response times.



**Figure 13.4** Servo Tune interface.

Manual tuning of motors is divided into the following sections:

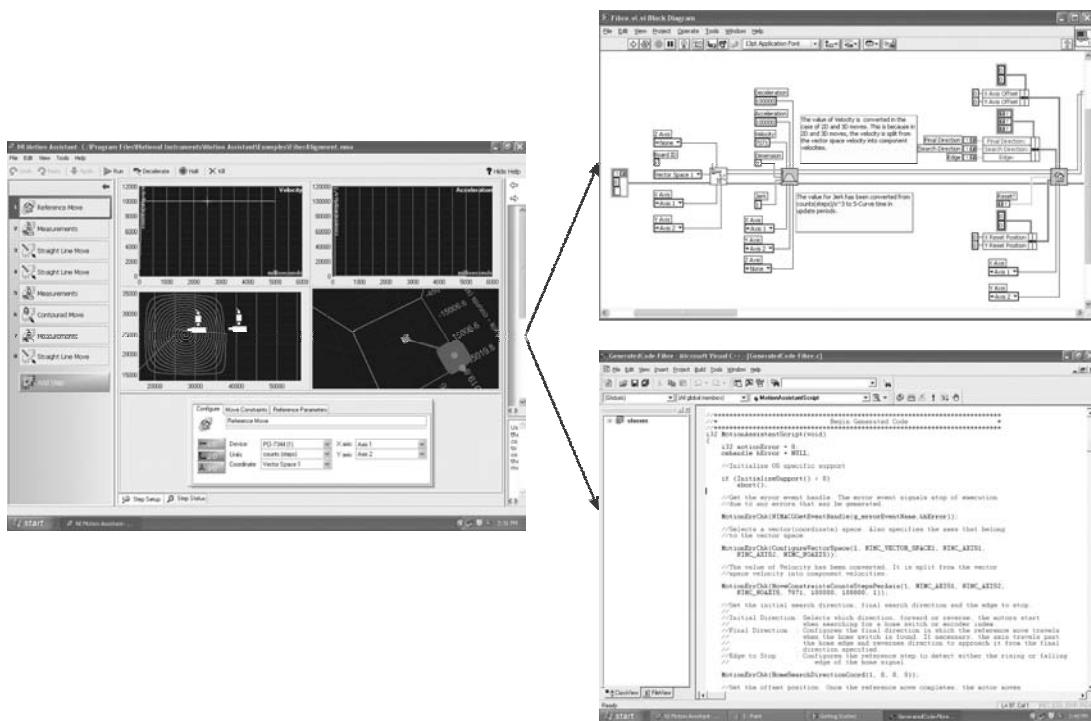
- Tuning the System
- Proportional Gain ( $K_p$ )
- Derivative Gain ( $K_d$ )
- Integral Gain ( $K_i$ )
- Derivative Sampling Period ( $T_d$ )
- Understanding PID Parameters

For tuning the system use the *Control Loop* tab to view and edit the PID parameters. Auto Tune provides a tuned system, but for an optimally tuned system, it is necessary to fine-tune the final PID parameters. It may also be necessary to alter the PID parameters, depending on your specific circumstances. Advanced Tuning Techniques are divided into Bode Plots, Stability in the Frequency Domain and Advanced Control Loop Parameters. Bode plots are the frequency response of your system. Stability in the Frequency Domain uses Bode plots to measure system stability. Advanced Control Loop Parameters are often necessary when using velocity or voltage amplifiers.

### 13.2.2 Prototyping

When you have configured your system, you can start prototyping and developing your application. In this phase, you create your motion control profiles and test them on your system to make sure they are what you intended. For prototyping, National Instruments offers a tool called NI Motion Assistant. NI Motion Assistant is an interactive tool with which you can configure moves using a point-and-click environment and generate LabVIEW code based on the moves you configure. The

key benefit of NI Motion Assistant lies in the difference between configurable and programmable environments. With configurable environments, you can start your development without programming. You can think of the tasks available in NI Motion Assistant as prewritten blocks of code that you simply configure to meet your needs. Programmable environments, on the other hand, require you to use standard programming languages such as LabVIEW, C or Visual Basic to accomplish your tasks. Unfortunately, many configurable environments may be limited in functionality or in the ability to integrate with other I/O outside of motion. NI Motion Assistant bridges the gap between programmable and configurable environments by offering all configurable system features as well as LabVIEW code generation. In Figure 13.5 NI Motion Assistant helps you quickly prototype your application and then convert your project into LabVIEW VIs or C code for further development.



**Figure 13.5** NI Motion Assistant.

### 13.2.3 Development

After the prototyping phase, the next step is to develop the final application code. For this, you use driver-level software in an ADE such as LabVIEW, C or Visual Basic. For a National Instruments motion controller, you use NI-Motion driver software. The NI-Motion driver software contains functions with which you can communicate with NI motion controllers in Windows or LabVIEW Real-Time. NI-Motion also includes Measurement and Automation Explorer to help you easily configure and tune your motion system. For non-Windows systems, you can develop your own driver using the Motion Control Hardware DDK manual. It explains how to communicate on a low level with NI motion controllers. If you do not have the expertise or time to develop your own

driver, National Instruments Alliance Partner, Sensing Systems, offers a Linux and VxWorks driver, and can create drivers for other OSs, such as Mac OS X or RTX.

### 13.3 MOTION CONTROLLER

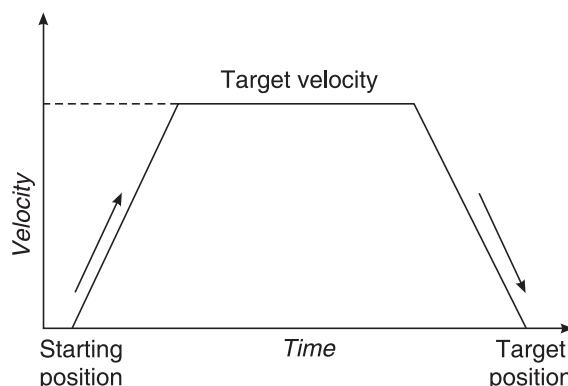
A motion controller acts as the brain of the motion control system and calculates each commanded move trajectory. Because this task is vital, it often takes place on a digital signal processor (DSP) on the board itself to prevent host-computer interference (you would not want your motion to stop because your antivirus software starts running). The motion controller uses the trajectories. It calculates to determine the proper torque command to send to the motor amplifier and actually causes motion.

The motion controller must also close the PID control loop. Because this requires a high level of determinism and is vital to consistent operation, the control loop typically closes on the board itself. Along with closing the control loop, the motion controller also manages supervisory control by monitoring the limits and emergency stops to ensure safe operation. Directing each of these operations to occur on the board or in a real-time system ensures the high reliability, determinism, stability and safety necessary to create a working motion control system.

#### 13.3.1 Calculating the Trajectory

The motion trajectory describes the motion controller board control or command signal output to the driver/amplifier, resulting in a motor/motion action that follows the profile. The typical motion controller calculates the motion profile trajectory segments based on the parameter values you program. The motion controller uses the desired target position, maximum target velocity and acceleration values you give it to determine how much time it spends in the three primary move segments (which include acceleration, constant velocity and deceleration).

For the acceleration segment of a typical trapezoidal profile as in Figure 13.6, motion begins from a stopped position or previous move and follows a prescribed acceleration ramp until the speed reaches the target velocity for the move. Motion continues at the target velocity for a prescribed period until the controller determines that it is time to begin the deceleration segment and slows the motion to a stop exactly at the desired target position.



**Figure 13.6** A typical trapezoidal velocity profile.

If a move is short enough that the deceleration beginning point occurs before the acceleration has completed, then the profile appears triangular instead of trapezoidal and the actual velocity attained may fall short of the desired target velocity. S-curve acceleration/deceleration is a basic trapezoidal trajectory enhancement where the acceleration and deceleration ramps are modified into a nonlinear, curved profile. This fine control over ramp shape is very useful for tailoring motion trajectory performance based upon the inertial, frictional forces, motor dynamics and other mechanical motion system limitations.

### 13.3.2 Selecting the Right Motion Controller

NI offers three main families of DSP-based motion controllers, including the low-cost NI 733x series, the mid-range NI 734x series and the high-performance NI 735x series. The NI 733x series low-cost controllers offer four-axis stepper motor control and most of the basic functions you need for a wide variety of applications, including single and multiaxis point-to-point motion. The NI 734x series is the mid-range series that offers up to four axes of both stepper and servo control, as well as some higher-performance features such as contouring and electronic gearing. The NI 735x series is the most advanced series that offers up to eight axes of stepper and servo control, extra I/O, and many powerful features including sinusoidal commutation for brushless motors and 4 MHz periodic breakpoints (or position triggers) for high-speed integration.

### 13.3.3 Creating Custom Motion Controllers

While current motion controllers with DSPs are suitable for many applications, when it comes to high-precision motion control with servo update rates as fast as 200 kHz, machine builders turn to designing their own motion controllers on a custom PCB. Not only is the development expensive in terms of time and cost, but the fixed personality of the motion controller makes the system inflexible for future redesigns or for accommodating variations in the motion control algorithms at run-time. Some applications that need such a high level of precision and flexibility include wafer processing machines in the semiconductor industry, or in-line vehicle sequencing (ILVS) reconfigurable-at-run-time assembly line for the automotive industry. National Instruments' reconfigurable I/O (RIO) technology coupled with NI SoftMotion technology provides the right tools for machine builders who want high-precision customized motion control with the complete flexibility of an FPGA. In addition to high-precision applications, machine builders and OEMs also can use the NI SoftMotion Development Module to implement multiaxis coordinated motion control using NI LabVIEW on a variety of platforms—from NI plug-in M Series DAQ devices for industrial PCs and PXI to rugged systems using NI CompactRIO and NI Compact FieldPoint programmable automation controllers (PACs).

## 13.4 MOVE TYPES

### 13.4.1 Single-Axis, Point-to-Point Motion

One of the most commonly used profiles is the simple, single-axis, point-to-point move, which requires the position to which the axis needs to move. Often, it also requires the velocity and acceleration (usually supplied by a default setting) at which you want the motion to move. Figure 13.7 shows a Single-axis, point-to-point motion in LabVIEW.

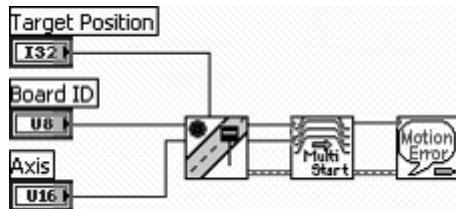


Figure 13.7 Single-axis, point-to-point motion.

### 13.4.2 Coordinated Multi-Axis Motion

Another type of motion is coordinated multiaxis motion or vector motion. This move is often point-to-point motion, but in 2D or 3D space. Vector moves require the final positions on the X, Y, and/or Z axes. Your motion controller also requires some type of vector velocity and acceleration. This motion profile is commonly found in XY-type applications such as scanning or automated microscopy. Figure 13.8 shows how to accomplish a two-axis move using LabVIEW. For more information on coordinated motion, view the examples in the LabVIEW Multiaxis.llb library in NI-Motion driver software.

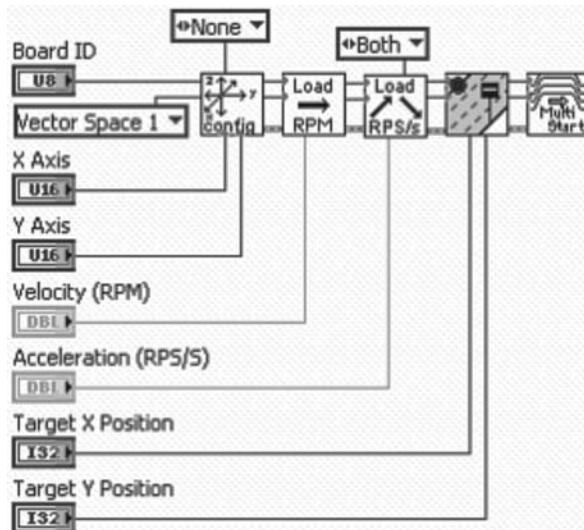
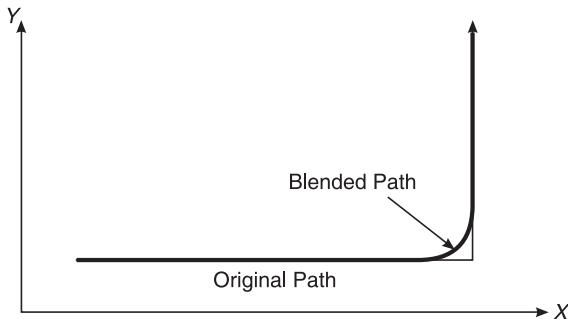


Figure 13.8 Coordinated multi-axis motion.

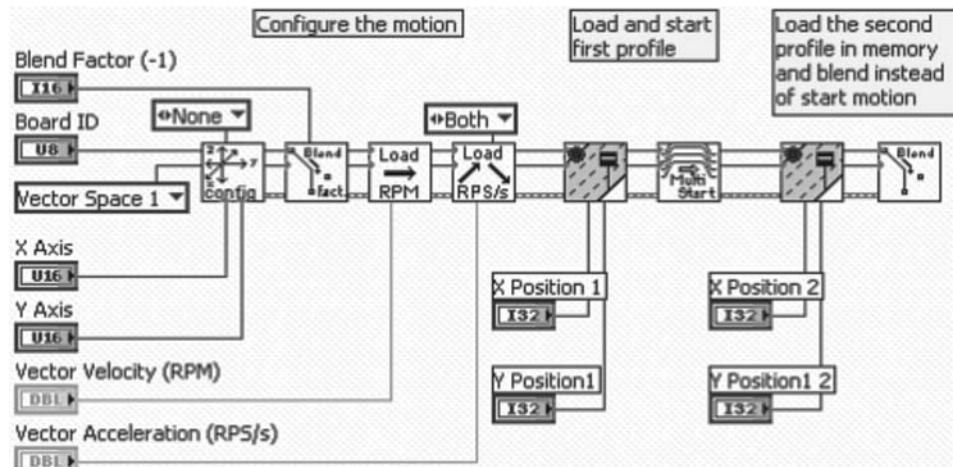
### 13.4.3 Blended Motion

Blended motion shown in Figure 13.9 involves two moves fused together by a blend that causes the moves to act as one. Blended moves require two moves and a blend factor that specifies the blend size. Blending is useful for applications requiring continuous motion between two different moves. However, in blended motion, your system does not pass through all of the points in your original trajectory. If the specific position along the path is important to you, consider a contouring motion. Figure 13.10 explains blending between two vector moves in LabVIEW. For more

information on blending, view the Sequence of Blended Vector Moves example program in NI-Motion driver software.



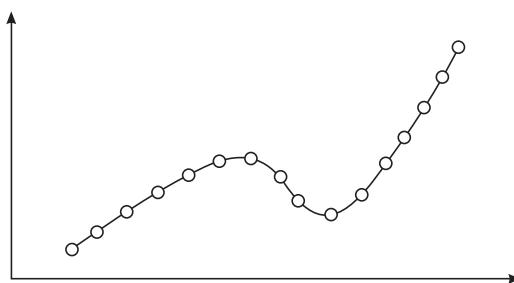
**Figure 13.9** Blended motion.



**Figure 13.10** Blended motion in LabVIEW.

#### 13.4.4 Contoured Motion

With contouring, you can supply a position buffer and create a smooth path or spline through them as shown in Figure 13.11. Contouring holds an advantage over blending in that it guarantees that



**Figure 13.11** Contoured motion.

the system passes through each position. Figure 13.12 explains a contoured move using LabVIEW. For more information on contouring, view the examples in the Countouring.llb example library found in NI-Motion driver software.

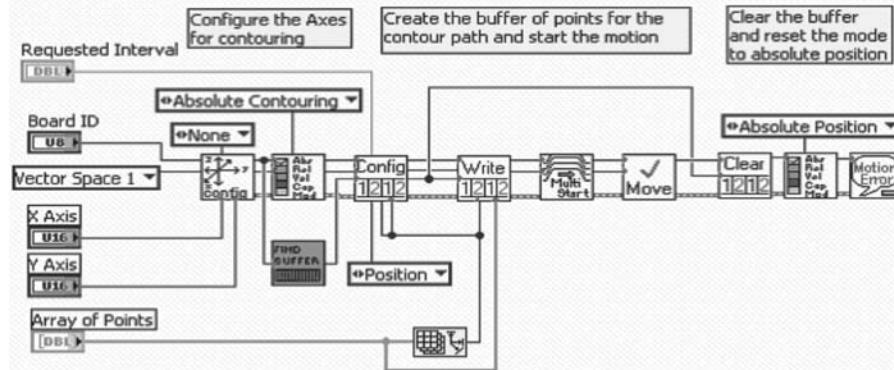


Figure 13.12 Contoured motion in LabVIEW.

### 13.4.5 Electronic Gearing

With electronic gearing, you can simulate the motion that would occur between two mating gears without using real gears. You use electronic gearing by supplying a gear ratio between a slave axis and a master axis, encoder, or ADC channel. Figure 13.13 shows how to configure a slave axis to follow a master axis.

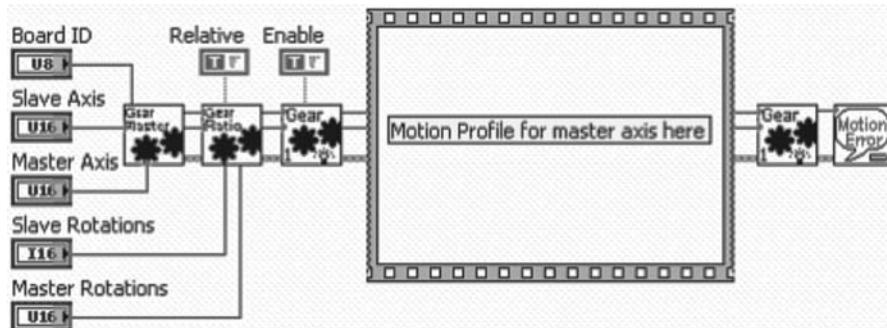


Figure 13.13 Electronic gearing in LabVIEW.

## 13.5 MOTOR AMPLIFIERS AND DRIVES

The motor amplifier or drive is the part of the system that takes commands from the motion controller in the form of analog voltage signals with low current and converts them into signals with high current to drive the motor. Motor drives come in many different varieties and are matched to the specific type of motor they drive. For example, a stepper motor drive connects to stepper motors, and not servomotors. Along with matching the motor technology, the drive must also provide the correct peak current, continuous current and voltage to drive the motor. If your drive supplies too

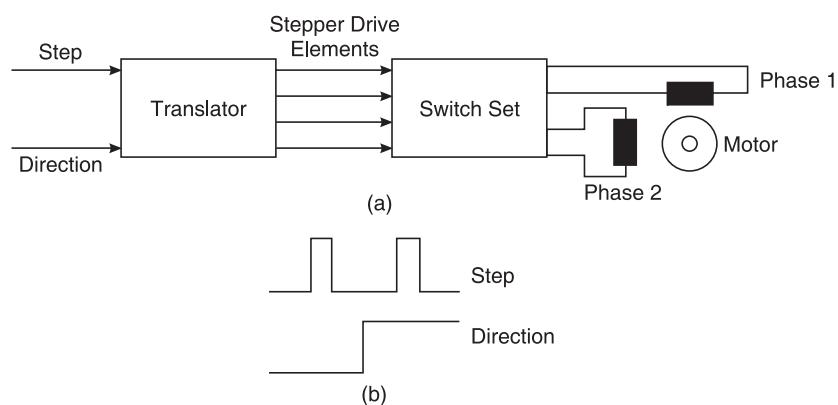
much current, you risk damaging your motor. If your drive supplies too little current, your motor does not reach full torque capacity. If your voltage is too low, your motor cannot run at its full speed.

### 13.5.1 Simple Servo Amplifiers

The output stage of all servo amplifiers is an analog circuit. The analog circuit provides a means to allow the voltage and current for the motor to be adjusted to control position, velocity and torque. The feedback and comparator stages can be any mixture of digital and analog devices. For example, if the feedback section uses a resolver, the output of this device is analog, so the section it works with is generally also analog. If the feedback device is an encoder, its output is digital, and the digital signal can be converted through a frequency-to-voltage converter so that the signal is usable in an analog circuit. Or it can be filtered and can use a digital value. The advent of microprocessors has allowed the digital values to be used through every part of the servo controller except the final output stage.

### 13.5.2 Stepper Motor Amplifiers

The stepper motor system consists of a translator circuit that receives a signal which includes the number of steps and the direction. The translator circuit sends four individual control signals to the switch set circuit, and the switch set circuit sends power signals to each of the two phases (windings) in the stepper motor. Figure 13.14(a) shows a diagram of the typical stepper motor system. The signal for the number of steps is included with this diagram and it is a series of square wave pulses (one for each step the rotor should move). The signal for the direction is a constant-voltage signal that is either positive or negative as shown in Figure 13.14(b). The translator typically receives its signal from a programmable logic controller (PLC) or other type of microprocessor controller. In some systems the controller is specifically designed to provide motion control or sequential control. The controller sends a command signal that consists of the number of steps the rotor should turn, and the direction signal indicates the direction. The step signals can be detected with LEDs or with an oscilloscope as they are sent to the translator. This means that if you are troubleshooting the translator and you want to know if it is receiving the



**Figure 13.14** (a) A translator and stepper motor and (b) stepper signal and direction signal.

pulses that represent the steps, you can use an LED indicator or scope to see these pulses. The direction of rotation signal can be detected with a voltmeter to determine if the signal is a positive voltage or negative voltage.

The simplest type of drive amplifier for a stepper motor is the *unipolar drive*. This type of amplifier is called a unipolar drive because current can only flow in one direction at any one time. The motor must be a *bipolar type* so that current can be reversed in the second segment of the winding to get the motor to run in the reverse direction. The chopper drive amplifier provides a means to control the current in the stepper windings to provide better torque control.

### 13.5.3 AC Servo Amplifiers

The amplifier for AC three-phase motors includes a pulse-width modulation circuit for voltage, current and frequency control. This type of motor is also called a *star connection* when it is used with brushless AC servomotors. This amplifier has a velocity amplifier that receives the original command signal for the amplifier and the velocity feedback. The op amp provides an output that represents the difference (error) between the command signal and the feedback signal. The output of the velocity amp is sent to the torque amp, where it is combined with the feedback from the current-sensing block. The output from this op amp is sent to the logic and PWM circuit block where it acts as the command signal. The position encoder provides the feedback signal for this block. This means that the velocity and position amplifiers are actually a closed-loop system within a closed-loop system. The gain for each of these amplifiers must be tuned so that the system has the best torque response and smooth acceleration and deceleration.

The feedback mechanism is generally a brushless DC tach generator, or an AC generator. Each of these feedback mechanisms provides smooth feedback voltages. If an encoder is used, its binary (digital) signal must be converted to an analog signal through a D/A converter or a frequency-to-analog F/A type converter if the signal is produced as a frequency.

### 13.5.4 DC Servo Amplifiers

The amplifiers for DC servomotors are slightly different from the push-pull amplifier and the chopper amplifier in that the power transistors can have a constant bias on their base rather than a pulsed signal. The power supply for this amplifier is AC voltage. The first part of this circuit is the bridge rectifier that provides a DC voltage at the DC bus. The output stage of this amplifier uses two transistors and two capacitors that are connected across the DC motor armature. The base of each of the power transistors is controlled by a switching circuit. This circuit can be controlled by an analog circuit or from a microprocessor. One of the drawbacks of a two-transistor amplifier is that the transistors must handle large amounts of current.

## 13.6 MOTOR FUNDAMENTALS

Motors come in many different types, shapes, and sizes. Most of the motors used in motion control can be divided into two categories: servomotors and stepper motors.

### 13.6.1 Servomotors

One of the main differences between servomotors and stepper motors is that servomotors, by definition, run using a control loop and require feedback of some kind. A control loop uses feedback from the motor to help the motor get to a desired state (position, velocity, and so on). There are many different types of control loops. Generally, the PID (Proportional, Integral, Derivative) control loop is used for servomotors. When using a control loop such as PID, you may need to tune the servomotor. Tuning is the process of making a motor respond in a desirable way. Tuning a motor can be a very difficult and tedious process, but is also an advantage in that it lets the user have more control over the behavior of the motor. Since servomotors have a control loop to check what state they are in, they are generally more reliable than stepper motors. When a stepper motor misses a step for any reason, there is no control loop to compensate in the move. The control loop in a servomotor is constantly checking to see if the motor is on the right path and, if it is not, it makes the necessary adjustments. In general, servomotors run more smoothly than stepper motors except when microstepping is used. Also, as speed increases, the torque of the servo remains constant, making it better than the stepper at high speeds (usually above 1000 RPM). For information about how servomotors work see the related link below.

Some of the advantages of servomotors over stepper motors are high intermittent torque, high torque to inertia ratio, high speeds, work well for velocity control, available in all sizes and quiet. Some of the disadvantages of servomotors compared with stepper motors are more expensive than stepper motors, cannot work in an open loop, feedback is required, require tuning of control loop parameters and more maintenance due to brushes on brushed DC motors.

Servomotors are available as AC or DC motors. Early servomotors were generally DC motors because the only type of control for large currents was through SCRs for many years. As transistors became capable of controlling larger currents and switching the large currents at higher frequencies, the AC servomotor became used more often. Early servomotors were specifically designed for servo amplifiers. Today a class of motors is designed for applications that may use a servo amplifier or a variable-frequency controller, which means that a motor may be used in a servo system in one application and used in a variable-frequency drive in another application. Some companies also call any closed-loop system that does not use a stepper motor a servo system, so it is possible for a simple AC induction motor that is connected to a velocity controller to be called a servomotor.

Some changes that must be made to any motor that is designed as a servomotor includes the ability to operate at a range of speeds without overheating, the ability to operate at zero speed and retain sufficient torque to hold a load in position, and the ability to operate at very low speeds for long periods of time without overheating. Older-type motors have cooling fans that are connected directly to the motor shaft. When the motor runs at slow speed, the fan does not move enough air to cool the motor. Newer motors have a separate fan mounted so it will provide optimum cooling air. This fan is powered by a constant voltage source so that it will turn at maximum RPM at all times regardless of the speed of the servomotor. One of the most usable types of motors in servo systems is the permanent magnet (PM) type motor. The voltage for the field winding of the permanent magnet type motor can be AC voltage or DC voltage. Figure 13.15 is a cutaway picture of a PM motor showing the housing, rotor and stator. The major difference with this type of motor is that it may have gear reduction to be able to move larger loads quickly

from a stand still position. This type of PM motor also has an encoder or resolver built into the motor housing. This ensures that the device will accurately indicate the position or velocity of the motor shaft.

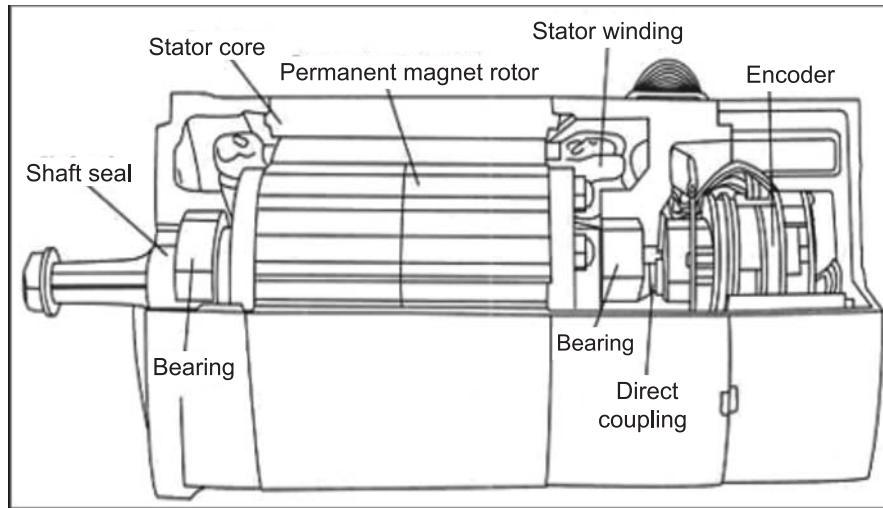
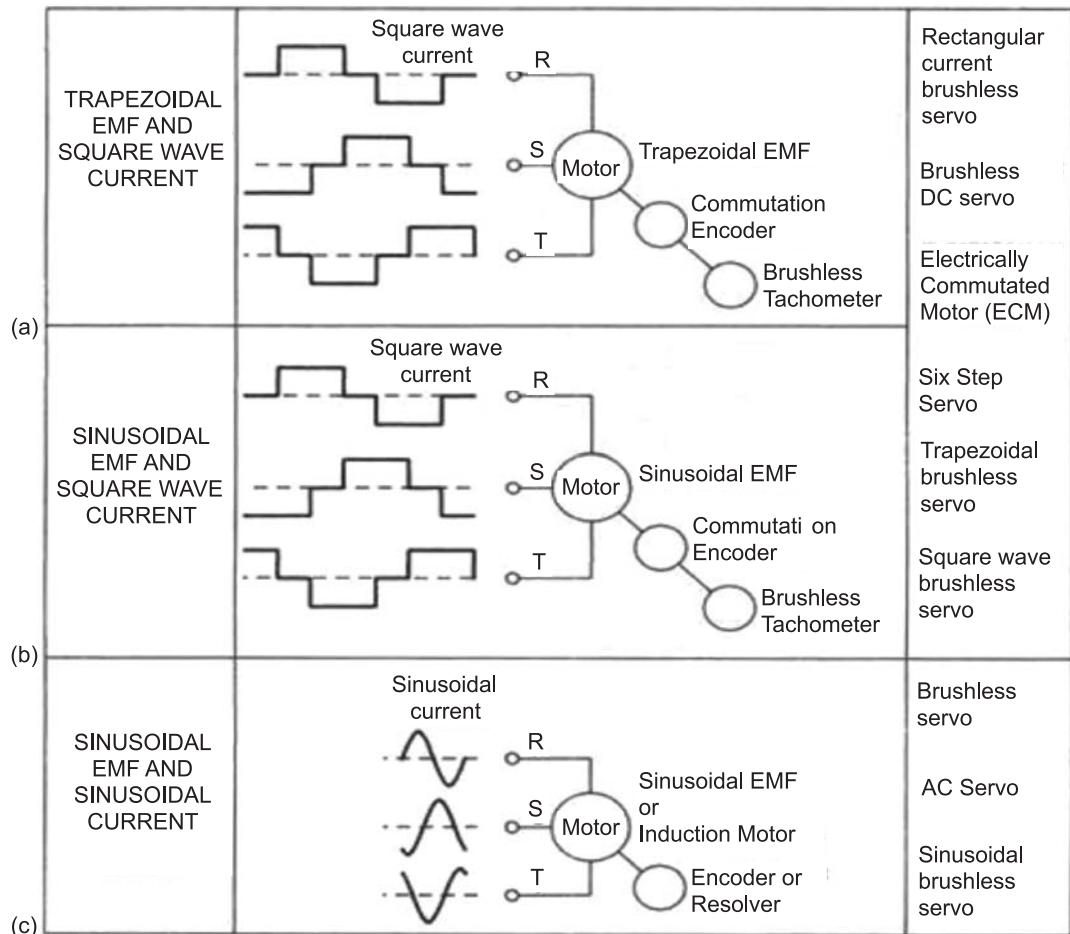


Figure 13.15 Typical PM servomotors.

### 13.6.2 Brushless Servomotors

The brushless servomotor is designed to operate without brushes. This means that the commutation that the brushes provided must now be provided electronically. Electronic commutation is provided by switching transistors on and off at appropriate times. Figure 13.16 shows three examples of the voltage and current waveforms that are sent to the brushless servomotor. Figure 13.17 shows an example of the three windings of the brushless servomotor. The main point about the brushless servomotor is that it can be powered by either AC voltage or DC voltage. Figure 13.16 shows three types of voltage waveforms that can be used to power the brushless servomotor.

Figure 13.16(a) shows a trapezoidal EMF (voltage) input and a square wave current input. Figure 13.16(b) shows a sinusoidal waveform for the input voltage and a square wave current waveform. Figure 13.16(c) shows a sinusoidal input waveform and a sinusoidal current waveform. This has become the most popular type of brushless servomotor control. The sinusoidal input and sinusoidal current waveform are the most popular voltage supplies for the brushless servomotor. Figure 13.17 shows three sets of transistors that are similar to the transistors in the output stage of the variable-frequency drive. In Figure 13.17(a) the transistors are connected to the three windings of the motor in a similar manner as in the variable-frequency drive. In Figure 13.17(b) the diagram of the waveforms for the output of the transistors is shown as three separate sinusoidal waves. The waveforms for the control circuit for the base of each transistor are shown in Figure(c). Figure 13.17(d) shows the back EMF for the drive waveforms.

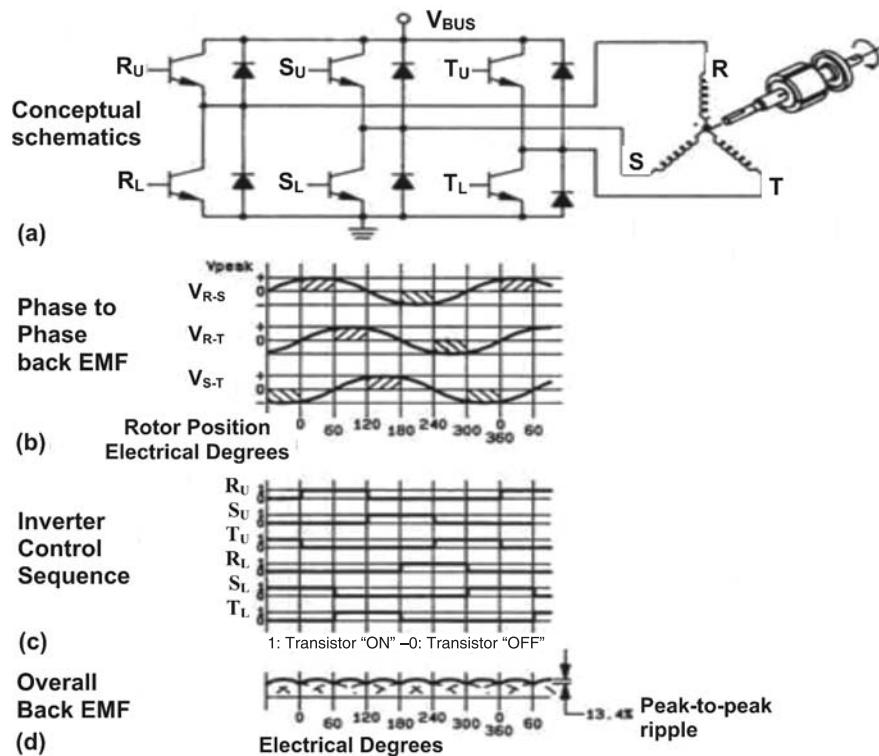


**Figure 13.16** (a) Trapezoidal input voltage and square wave current waveforms, (b) Sinusoidal input voltage and square wave output voltage waveforms and (c) Sinusoidal input voltage and sinusoidal current waveforms.

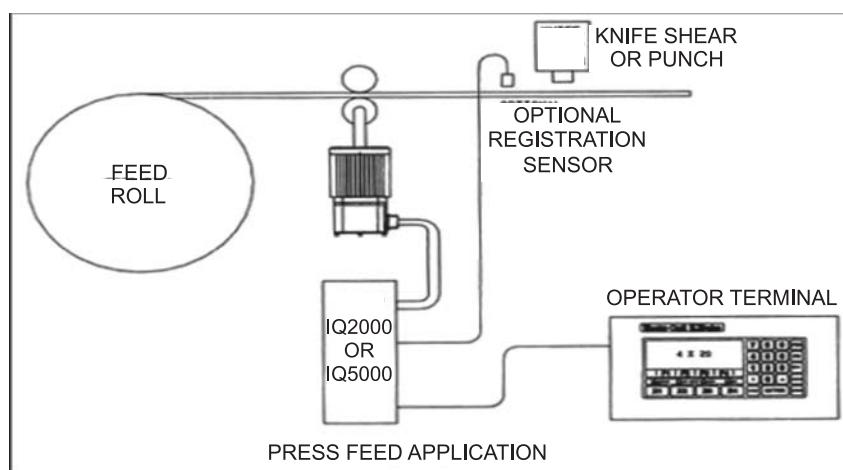
### 13.6.3 Servomotor Applications

Servomotors find application in press feed, in-line bottle filling, precision auger filling system, label applications and random timing infeed system.

Figure 13.18 shows an application of a servomotor controlling the speed of material as it enters a press for cutting pieces to size. In this application, sheet material is fed into a press where it is cut off to length with a knife blade or sheer. The sheet material may have a logo or other advertisement that must line up registration marks with the cut-off point. In this application the speed and position of the sheet material must be synchronized with the correct cut-off point. The feedback sensor could be an encoder or resolver that is coupled with a photoelectric sensor to determine the location of the registration mark. An operator panel is provided so that the operator



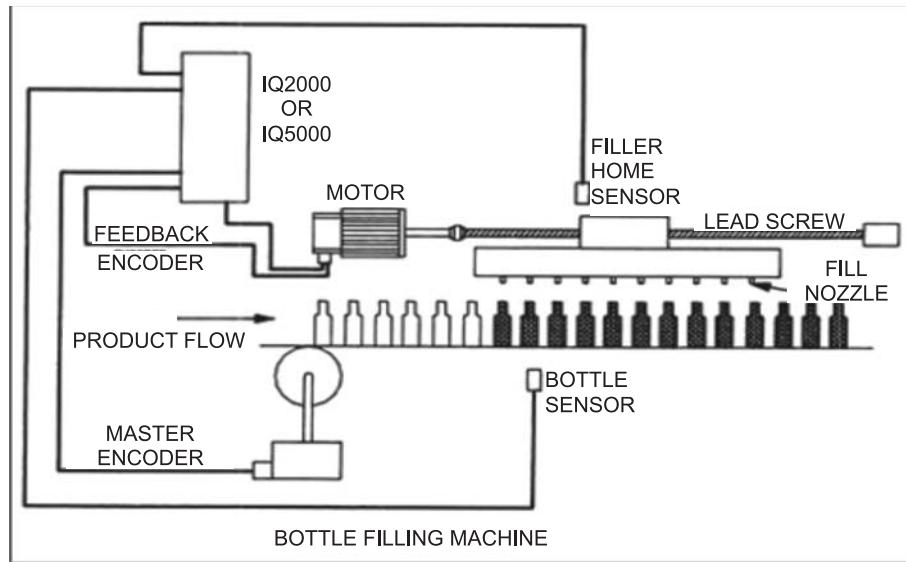
**Figure 13.17** (a) Transistors connected to the three windings of the brushless servomotor, (b) Waveforms of the three separate voltages that are used to power the three motor windings, (c) Waveforms of the signals used to control the transistor sequence and (d) Waveform of the overall back EMF.



**Figure 13.18** Servomotor used to control a press feed.

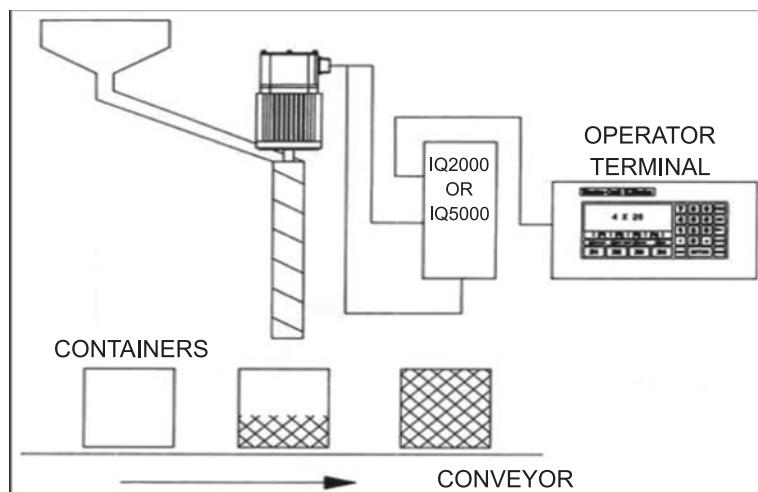
can jog the system for maintenance of the blades, or when loading a new roll of material. The operator panel could also be used to call up parameters for the drive that correspond to each type of material that is used. The system could also be integrated with a programmable controller or other type of controller and the operator panel could be used to select the correct cutoff points for each type of material or product that is run.

A second application is In-line Bottle Filling as shown in Figure 13.19. In this application multiple filling heads line up with bottles as they move along a continuous line. Each of the filling heads must match up with a bottle and track the bottle while it is moving. Product is dispensed as the nozzles move with the bottles. In this application 10 nozzles are mounted on a carriage that is driven by a ball-screw mechanism. The ball-screw mechanism is also called a lead screw. When the motor turns the shaft of the ball screw, the carriage will move horizontally along the length of the ball-screw shaft. This movement will be smooth so that each of the nozzles can dispense product into the bottles with little spillage. The servo drive system utilizes a positioning drive controller with software that allows the position and velocity to be tracked as the conveyor line moves the bottles. A master encoder tracks the bottles as they move along the conveyor line. An auger feed system is also used just prior to the point where the bottles enter the filling station. The auger causes a specific amount of space to be set between each bottle as it enters the filling station. The bottles may be packed tightly as they approach the auger, but as they pass through the auger their space is set exactly so that the necks of the bottles will match the spacing of the filling nozzles. A detector is also in conjunction with the dispensing system to ensure that no product is dispensed from a nozzle if a bottle is missing or large spaces appear between bottles. The servo drive system compares the position of the bottles from the master encoder to the feedback signal that indicates the position of the filling carriage that is mounted to the ball screw. The servo drive amplifier will increase or decrease the speed of the ball-screw mechanism so that the nozzles will match the speed of the bottles exactly.



**Figure 13.19** Beverage-filling station controlled by a servomotor.

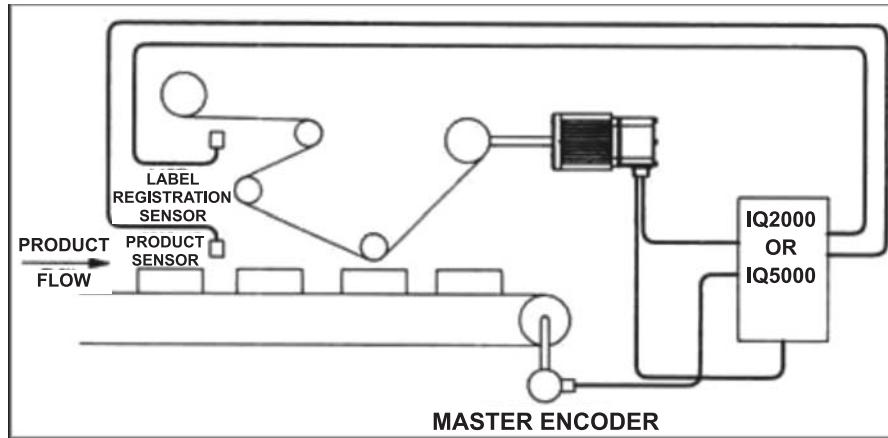
A third application for a servosystem shown in Figure 13.20 is for precision auger filling system. In this application a large filling tank is used to fill containers as they pass along a conveyor line. The material that is dispensed into the containers can be a single material fill or it can be one of several materials added to a container that is dumped into a mixer for a blending operation. Since the amount of material that is dispensed into the container must be accurately weighed and metered into the box, an auger that is controlled by a servo system is used. The feedback sensor for this system can be a weighing system such as the load cell discussed in earlier chapters. The command signal can come from a programmable controller or the operator can enter it manually by selecting a recipe from the operator's terminal. The amount of material can be different from recipe to recipe. The speed of the auger can be adjusted so that it runs at high speed when the container is first filled, and the speed can be slowed to a point where the final grams of material can be metered precisely as the container is filled to the proper point. As the price of material increases, precision filling equipment can provide savings as well as quality in the amount of product used in the recipe.



**Figure 13.20** Application of a precision auger filling station controlled by a servomotor.

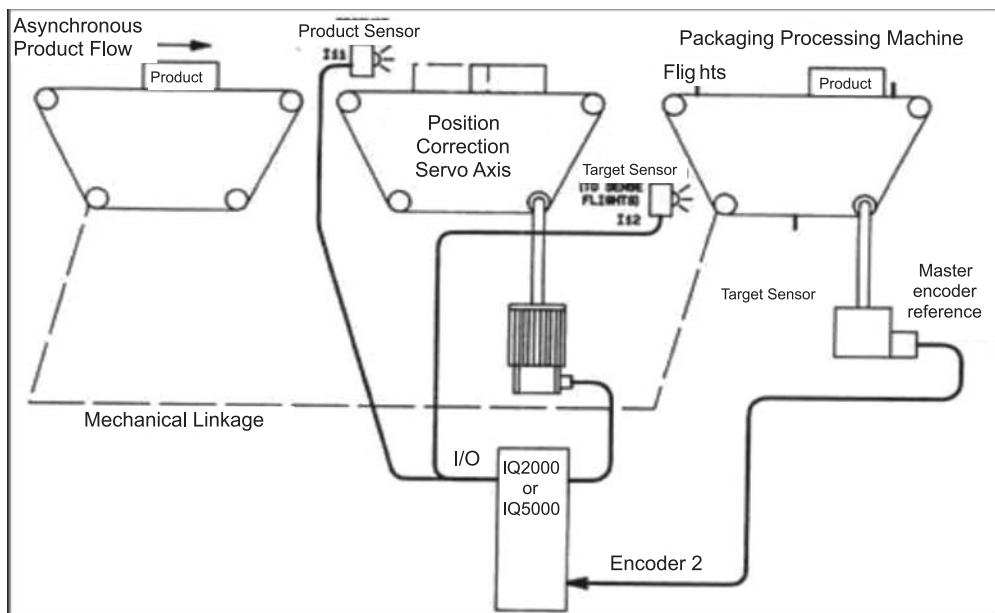
The fourth application has a servomotor controlling the speed of a label-feed mechanism that pulls preprinted labels from a roll and applies them to packages as they move on a continuous conveyor system past the labeling mechanism. The feedback signals are provided by an encoder that indicates the location of the conveyor, tach generator that indicates the speed of the conveyor, and a sensor that indicates the registration mark on each label. The servo positioning system is controlled by a microprocessor that sets the error signal, and the servo amplifier that provides power signals to the servomotor. This application is shown in Figure 13.21.

The fifth application in random timing infeed system is presented in Figure 13.22, and it shows a series of packaging equipment that operates as three separate machines. The timing cycle of each station of the packaging system is independent from the others. The packaging system consists of an *infeed conveyor*, a *positioning conveyor*, and a *wrapping station*. The infeed conveyor and the wrapping station are mechanically connected so that they run at the same speed. The position of the packages on the wrapping station must be strictly controlled so that the packages



**Figure 13.21** Labeling application controlled by a servomotor.

do not become too close to each other. A piece of metal called a *flight* is connected to the wrapping station conveyor at specific points to ensure each package stays in position. A sensor is mounted at the beginning of the positioning conveyor to determine the front edge of the package when it starts to move onto the positioning conveyor. A second sensor is positioned at the bottom of the packaging conveyor to detect the flights. Both of these signals from the sensors are sent to the servomotor to provide information so the servo can adjust the speed of the positioning conveyor so that each package aligns with one of the flights as it moves onto the packaging conveyor. This application shows that the servo positioning controller can handle a variety of different signals from more than one sensor because the controller uses a microprocessor.



**Figure 13.22** Packaging system with random timing functions controlled by a servomotor.

### 13.6.4 Stepper Motors

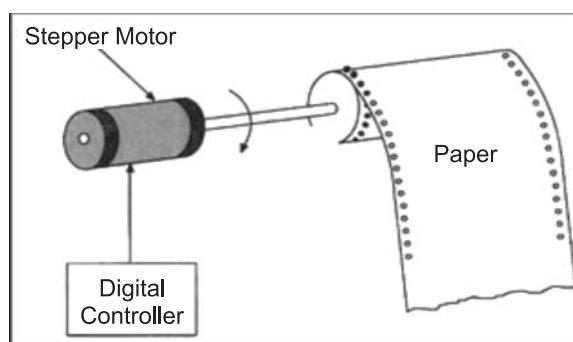
Stepper motors are less expensive and typically easier to use than a servo motor of a similar size. They are called stepper motors because they move in discrete steps. Controlling a stepper motor requires a stepper drive and a controller. You control a stepper motor by providing the drive with a step and direction signal. The drive then interprets these signals and drives the motor. Stepper motors can be run in an open loop configuration (no feedback) and are good for low-cost applications. In general, a stepper motor will have high torque at low speeds, but low torque at high speeds. Movement at low speeds is also choppy unless the drive has microstepping capability. At higher speeds, the stepper motor is not as choppy, but it does not have as much torque. When idle, a stepper motor has a higher holding torque than a servomotor of similar size, since current is continuously flowing in the stepper motor windings.

Some of the advantages of stepermotors over servomotors are low cost, can work in an open loop (no feedback required), excellent holding torque (eliminated brakes/clutches), excellent torque at low speeds, low maintenance (brushless), very rugged in any environment, excellent for precise positioning control and no tuning required.

Some of the disadvantages of stepper motors in comparison with servomotors are rough performance at low speeds unless you use microstepping, consume current regardless of load, limited sizes available, noisy, torque decreases with speed and stepper motors can stall or lose position running without a control loop.

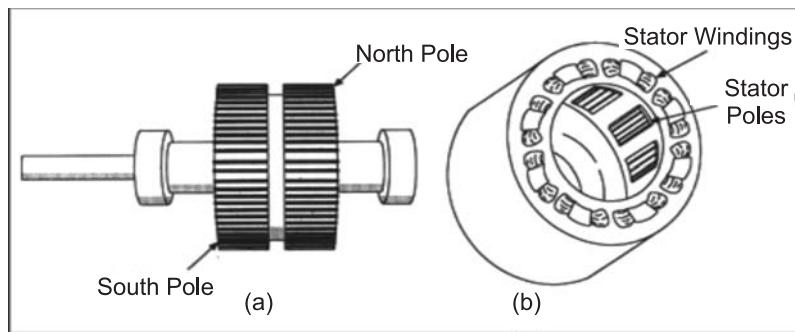
### 13.6.5 Stepper Motor Types

A stepper, or stepping motor converts electronic pulses into proportionate mechanical movement. Each revolution of the stepper motor's shaft is made up of a series of discrete individual steps. A step is defined as the angular rotation produced by the output shaft each time the motor receives a step pulse. These types of motors are very popular in digital control circuits, such as robotics, because they are ideally suited for receiving digital pulses for step control. Each step causes the shaft to rotate a certain number of degrees. A step angle represents the rotation of the output shaft caused by each step, measured in degrees. Figure 13.23 illustrates a simple application for a stepper motor. Each time the controller receives an input signal, the paper is driven a certain incremental distance. In addition to the paper drive mechanism in a printer, stepper motors are also popular in machine tools, process control systems, tape and disk drive systems and programmable controllers.



**Figure 13.23** Paper drive mechanism using stepper machine.

The most popular types of stepper motors are permanent-magnet (PM) and variable reluctance (VR) and Hybrid Stepper Motors. The *permanent-magnet stepper motor* operates on the reaction between a permanent-magnet rotor and an electromagnetic field. Figure 13.24 shows a basic two-pole PM stepper motor. The rotor shown in Figure 13.24(a) has a permanent magnet mounted at each end. The stator is illustrated in Figure 13.24(b). Both the stator and rotor are shown as having teeth. The teeth on the rotor surface and the stator pole faces are offset so that there will be only a limited number of rotor teeth aligning themselves with an energized stator pole. The number of teeth on the rotor and stator determine the step angle that will occur each time the polarity of the winding is reversed. The greater the number of teeth, the smaller the step angle.

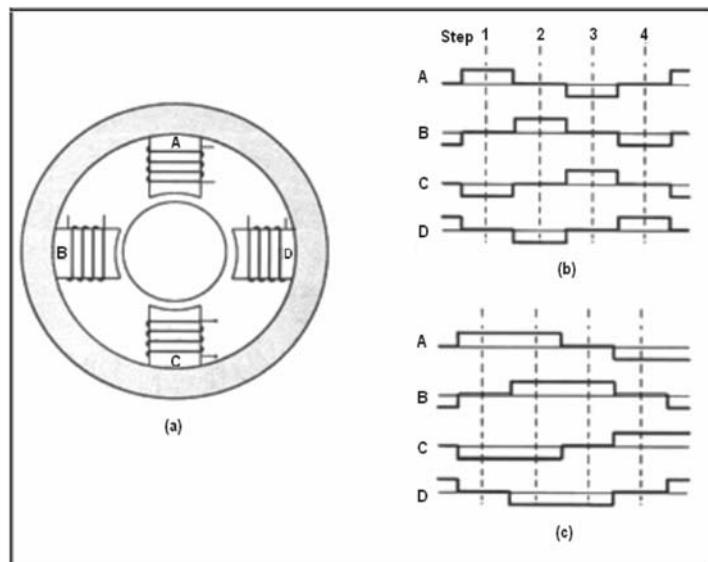


**Figure 13.24** Components of a PM stepper motor: (a) Rotor and (b) Stator.

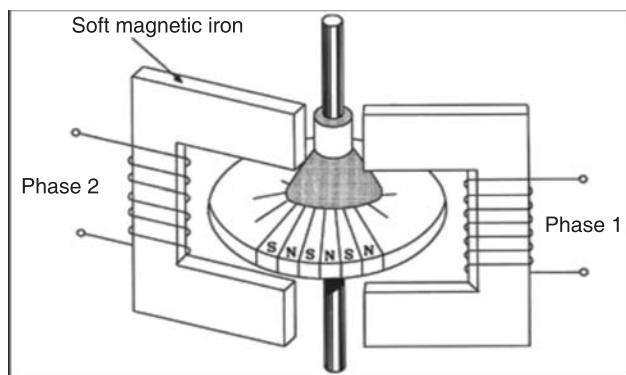
When a PM stepper motor has a steady DC signal applied to one stator winding, the rotor will overcome the residual torque and line up with that stator field. The holding torque is defined as the amount of torque required to move the rotor one full step with the stator energized. An important characteristic of the PM stepper motor is that it can maintain the holding torque indefinitely when the rotor is stopped. When no power is applied to the windings, a small magnetic force is developed between the permanent magnet and the stator. This magnetic force is called a residual, or detent torque. The detent torque can be noticed by turning a stepper motor by hand and is generally about one-tenth of the holding torque.

Figure 13.25(a) shows a permanent magnet stepper motor with four stator windings. By pulsing the stator coils in a desired sequence, it is possible to control the speed and direction of the motor. Figure 13.25(b) shows the timing diagram for the pulses required to rotate the PM stepper motor. This sequence of positive and negative pulses causes the motor shaft to rotate counterclockwise in 90° steps. The waveforms of Figure 13.25(c) illustrate how the pulses can be overlapped and the motor made to rotate counterclockwise at 45° intervals.

A more recent development in PM stepper motor technology is the thin-disk rotor. This type of stepper motor dissipates much less power in losses such as heat than the cylindrical rotor and as a result, it is considerably more efficient. Efficiency is a primary concern in industrial circuits such as robotics, because a highly efficient motor will run cooler and produce more torque or speed for its size. Thin-disk rotor PM stepper motors are also capable of producing almost double the steps per second of a conventional PM stepper motor. Figure 13.26 shows the basic construction of a thin-disk rotor PM motor. The rotor is constructed of a special type of cobalt-steel, and the stator poles are offset by one-half a rotor segment.



**Figure 13.25** (a) PM stepper motor, (b) 90° step and (c) 45° step.



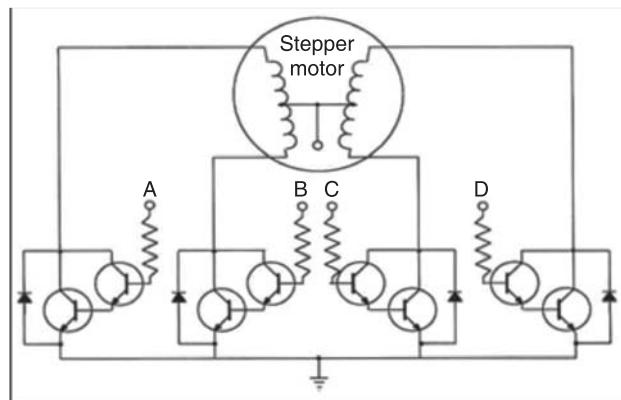
**Figure 13.26** Thin-disk rotor PM stepper motor.

The variable-reluctance (VR) stepper motor differs from the PM stepper in that it has no permanent-magnet rotor and no residual torque to hold the rotor at one position when turned off. When the stator coils are energized, the rotor teeth will align with the energized stator poles. This type of motor operates on the principle of minimizing the reluctance along the path of the applied magnetic field. By alternating the windings that are energized in the stator, the stator field changes, and the rotor is moved to a new position. The stator of a variable-reluctance stepper motor has a magnetic core constructed with a stack of steel laminations. The rotor is made of unmagnetized soft steel with teeth and slots.

The *hybrid step motor* consists of two pieces of soft iron, as well as an axially magnetized, round permanent-magnet rotor. The term *hybrid* is derived from the fact that the motor is operated under the combined principles of the permanent magnet and variable-reluctance stepper motors.

The stator core structure of a hybrid motor is essentially the same as its VR counterpart. The main difference is that in the VR motor, only one of the two coils of one phase is wound on one pole, while a typical hybrid motor will have coils of two different phases wound on one the same pole. The two coils at a pole are wound in a configuration known as a *bifilar* connection. Each pole of a hybrid motor is covered with uniformly spaced teeth made of soft steel. The teeth on the two sections of each pole are misaligned with each other by a half-tooth pitch. Torque is created in the hybrid motor by the interaction of the magnetic field of the permanent magnet and the magnetic field produced by the stator.

The direction of rotation is determined by applying the pulses to either the clockwise or counterclockwise drive circuits. Rotor displacement can be very accurately repeated with each succeeding pulse. Stepping motors are generally operated without feedback which simplifies the control circuit considerably. One of the most common stepper motor drive circuits is the *unipolar* drive shown in Figure 13.27. This circuit uses bifilar windings and four Darlington transistors to control the direction of rotation and the stepping rate of the motor.

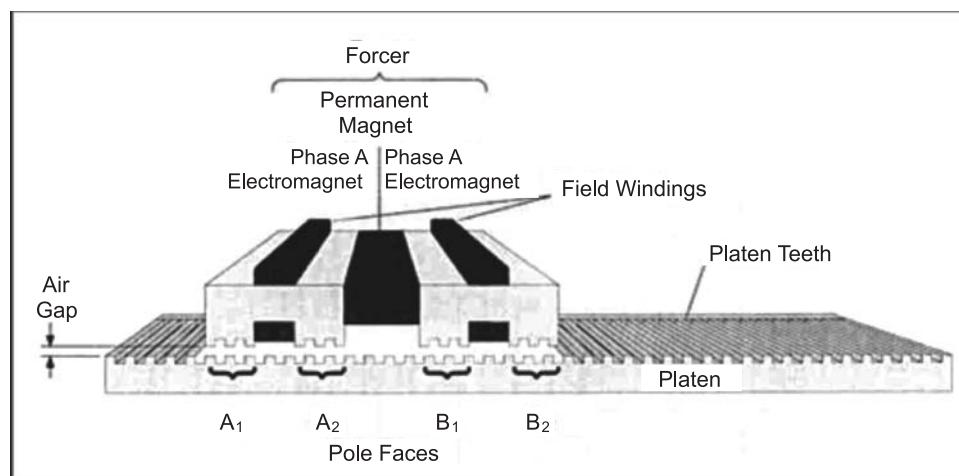


**Figure 13.27** Unipolar stepper motor drive.

### 13.6.6 Linear Stepper Motors

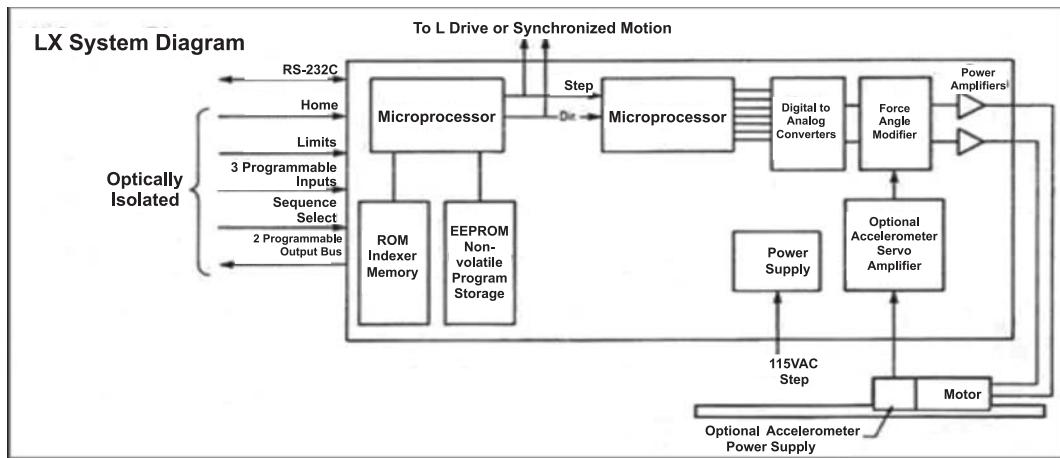
The linear stepper motor has been made flat instead of round, so its motion will be along a straight line instead of rotary. The basic parts of the linear motor are shown in Figure 13.28. The motor consists of a *platen* and a *forcer*. The forcer is shown on top of the platen of a linear motor and the electromagnets are identified on the forcer. The platen is the fixed part of the motor and its length will determine the distance the motor will travel. It has a number of teeth that are like the rotor in a traditional stepper motor except it is passive and is not a permanent magnet. The forcer consists of four pole pieces that each have three teeth. The pitch of each tooth is staggered with respect to the teeth of the platen. It uses mechanical roller bearings or air bearings to ride above the platen on an air gap so that the two never physically come into contact with each other. The magnetic field in the forcer is changed by passing current through its coils. This action causes the next set of teeth to align with the teeth on the platen and causes the forcer to move from tooth to tooth over the platen in linear travel. When the current pattern is reversed,

the forcer will reverse its direction of travel. A complete switching cycle consists of four full steps, which moves the forcer the distance of one tooth pitch over the platen. The typical resolution of a linear motor is 12,500 steps per inch, which provides a high degree of resolution. The typical load for a linear motor is low mass that requires high-speed movements. The forcer consists of two electromagnets (A and B) and one permanent magnet. The permanent magnet is a strong rare-earth permanent magnet. The electromagnets are formed in the shape of teeth so that their magnetic flux can be concentrated. In the diagram you can see that the forcer has four sets of teeth and these teeth are spaced in quadrature so that only one set of teeth is aligned with the teeth on the platen at any time. When current is applied to the coil (field winding) of the electromagnets, their magnetic flux passes through the air gap between the forcer and the platen, causing a strong attraction between the two. The magnetic flux from the electromagnets also tends to reinforce the flux lines of one of the permanent magnets and cancels the flux lines of the other permanent magnet. The attraction of the forces at the time when peak current is flowing is up to ten times the holding force. When a pattern of energizing one coil and then another is established, the resulting magnetic field will pull the motor in one direction from one tooth to the next. When current flow to the coil is stopped, the forcer will align itself to the appropriate tooth set and create a holding force that tends to keep the forcer from moving left or right to another tooth. The linear stepper motor controller sets the pattern for energizing and de-energizing the field coils so that the motor moves smoothly in either direction. By reversing the pattern, the direction the motor travels is reversed.



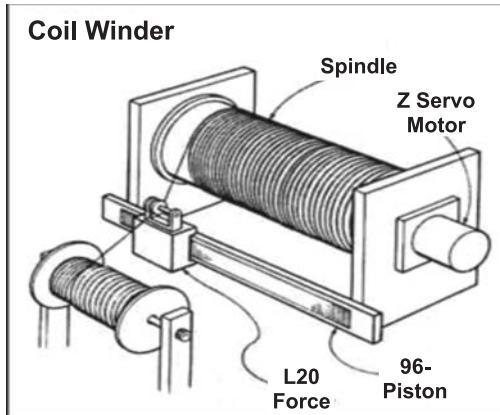
**Figure 13.28** Linear Stepper Motor.

Figure 13.29 shows a block diagram of the linear stepper motor controller. From this diagram you can see that it has a microprocessor that interfaces with a digital-to-analog converter, a force angle modifier, and a power amplifier. It also has a power supply for the amplifiers and it may have an accelerometer amplifier as an option. The microprocessor has ROM and EPROM memory to store programs.



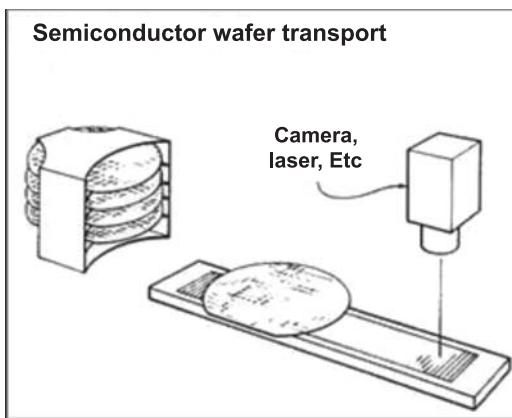
**Figure 13.29** Block diagram of a linear motor controller.

The applications for a linear motor tend to be straight-line motion. These types of applications are slightly different from traditional stepper motor applications where the rotary motion is converted to linear motion with a ball and screw, rack and pinion, or other method. Figure 13.30 shows the linear motor used in a coil winding positioner application to control the position of the coil winder. The linear motor in this application is teamed with a servomotor that controls the speed of the coil winding mechanism. The linear motor determines the exact location of the next coil that is added to the spool. The speed of the linear motor can be increased or decreased when the machine is spooling larger-diameter or smaller-diameter wire. The ability of the linear motor to provide small incremental steps makes it a good match for this application.



**Figure 13.30** Used in a coil winding application.

Figure 13.31 shows a second application where the linear motor is used to transport a semiconductor wafer through a precision laser inspection station. The linear motor provides excellent locating ability for this application. The linear motor acts as a transport for semiconductor wafers. The linear motor system offers increased throughput and gentle handling of the wafer.



**Figure 13.31** Used to transport a silicon semiconductor wafer through a laser inspection station.

### 13.6.7 Stepper Motor Applications

Stepper motors are used in a wide variety of applications in industry, including computer peripherals, business machines, motion control and robotics, which are included in process control and machine tool applications.

- Computer peripherals

<i>Application</i>	<i>Use</i>
floppy disc	position magnetic pickup
printer	carriage drive
printer	rotate character wheel
printer	paper feed
printer	ribbon wind/rewind
printer	position matrix print head
tape reader	index tape
plotter	X-Y-Z positioning
plotter	paper feed

- Business machines

<i>Application</i>	<i>Use</i>
card reader	position cards
copy machine	paper feed
banking systems	credit card positioning
banking systems	paper feed
typewriters (automatic)	head positioning
typewriters (automatic)	paper feed
copy machine	lens positioning
card sorter	route card flow

- **Process control**

<i>Application</i>	<i>Use</i>
carburetor adjusting	air-fuel mixture adjust
valve control	fluid gas metering
conveyor	main drive
in-process gaging	parts positioning
assembly lines	parts positioning
silicon processing	I.C. wafer slicing
IC bonding	chip positioning
laser trimming	X-Y positioning
liquid gasket dispensing	valve cover positioning
mail handling systems	feeding and positioning letters

- **Machine tool**

<i>Application</i>	<i>Use</i>
milling machines	X-Y-Z table positioning
drilling machines	X-Y table positioning
grinding machines	downfeed grinding wheel
grinding machines	automatic wheel dressing
electron beam welder	X-Y-Z positioning
laser cutting	X-Y-Z positioning
lathes	X-Y positioning
sewing	X-Y table positioning

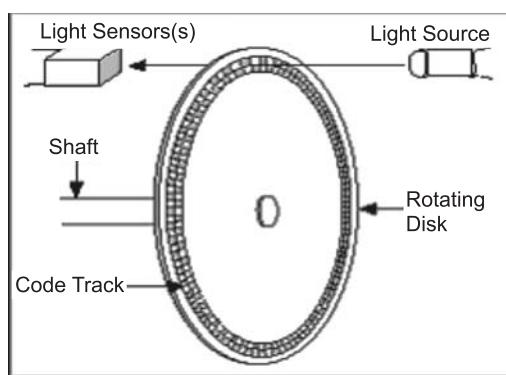
### 13.7 FEEDBACK DEVICES AND MOTION I/O

Feedback devices help the motion controller know the motor location. The most common position feedback device is the quadrature encoder, which gives positions relative to the starting point. Most motion controllers are designed to work with these types of encoders. Other feedback devices include potentiometers that give analog position feedback, tachometers that provide velocity feedback, absolute encoders for absolute position measurements, and resolvers that also give absolute position measurements. In any closed loop control system, a method for measuring the characteristic being controlled (i.e. position) needs to be implemented. This is known as *feedback*. Some of the different kinds of feedback used in motion control are relative position, absolute position, velocity and analog voltage feedback. For relative position feedback, the most common device used is the incremental encoder. This device increments as it turns and can be used to keep track of the total distance traveled from some point of reference. The most common absolute position feedback devices are absolute encoders and resolvers. An absolute encoder is similar to an incremental encoder except that it does not need to have a reference position and can know what position the shaft is in on start up. A resolver is also another absolute position device that is commonly used for

both position and velocity feedback. For velocity feedback, a tachometer is commonly used. Some applications require feedback other than the type that has been mentioned above. In these cases, analog feedback describing some characteristic of the system such as temperature or pressure can be useful in controlling motors. Common feedback devices are absolute encoders, linear and rotary encoders, resolvers and tachometers.

### 13.7.1 Encoders

An encoder is an electrical mechanical device that converts linear or rotary displacement into digital or pulse signals. The most popular type of encoder is the optical encoder, which consists of a rotating disk, a light source, and a photodetector (light sensor). The disk, which is mounted on the rotating shaft, has patterns of opaque and transparent sectors coded into the disk as in Figure 13.32. As the disk rotates, these patterns interrupt the light emitted onto the photodetector, generating a digital or pulse signal output.

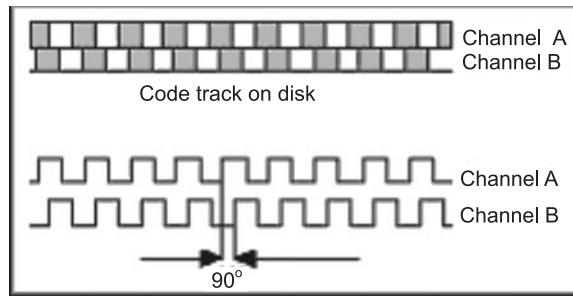


**Figure 13.32** Optical encoder.

There are two general types of encoders—absolute and incremental encoders. An absolute encoder generates a unique word pattern for every position of the shaft. The tracks of the absolute encoder disk, generally four to six, commonly are coded to generate binary code, binary-coded decimal (BCD), or gray code outputs. Absolute encoders are most commonly used in applications where the device will be inactive for long periods of time, there is risk of power down, or the starting position is unknown.

An incremental encoder generates a pulse, as opposed to an entire digital word, for each incremental step. Although the incremental encoder does not output absolute position, it does provide more resolution at a lower price. For example, an incremental encoder with a single code track, referred to as a *tachometer encoder*, generates a pulse signal whose frequency indicates the velocity of displacement. However, the output of the single-channel encoder does not indicate direction. To determine direction, a two-channel, or quadrature, encoder uses two detectors and two code tracks. The most common type of incremental encoder uses two output channels (A and B) to sense position. Using two code tracks with sectors positioned  $90^\circ$  out of phase as in Figure 13.33, the two output channels of the quadrature encoder indicate both position and direction of rotation. If A leads B, for example, the disk is rotating in a clockwise direction. If B leads A, then the disk is rotating in a counter-clockwise direction. Therefore, by monitoring both the number of pulses

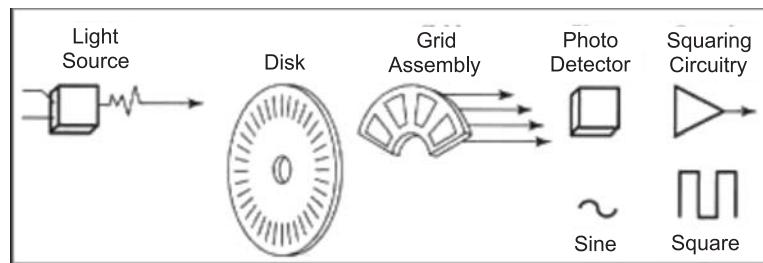
and the relative phase of signals A and B, you can track both the position and direction of rotation. In addition, some quadrature detectors include a third output channel, called a zero or reference signal, which supplies a single pulse per revolution. This single pulse can be used for precise determination of a reference position.



**Figure 13.33** Quadrature encoder output channels A and B.

### 13.7.2 Linear and Rotary Encoders

The linear and rotary encoder is an electrical mechanical device that can monitor motion or position. A typical encoder uses optical sensors to provide a series of pulses that can be translated into motion, position, or direction. Figure 13.34 shows a rotary encoder. The disk is very thin, and a stationary light-emitting diode (LED) is mounted so that its light will continually be focused through the glass disk. A light-activated transistor is mounted on the other side of the disk so that it can detect the light from the LED. The disk is mounted to the shaft of a motor or other device whose position is being sensed, so that when the shaft turns, the disk turns. When the disk lines up so the light from the LED is focused on the phototransistor, the phototransistor will go into saturation and an electrical square wave pulse will be produced. The square wave pulses are produced by the rotary encoder. This type of disk was used in early applications but the size of the holes in the metal disk limited the amount of accuracy that could be obtained. As more holes were cut in the disk, it became too fragile for industrial use.

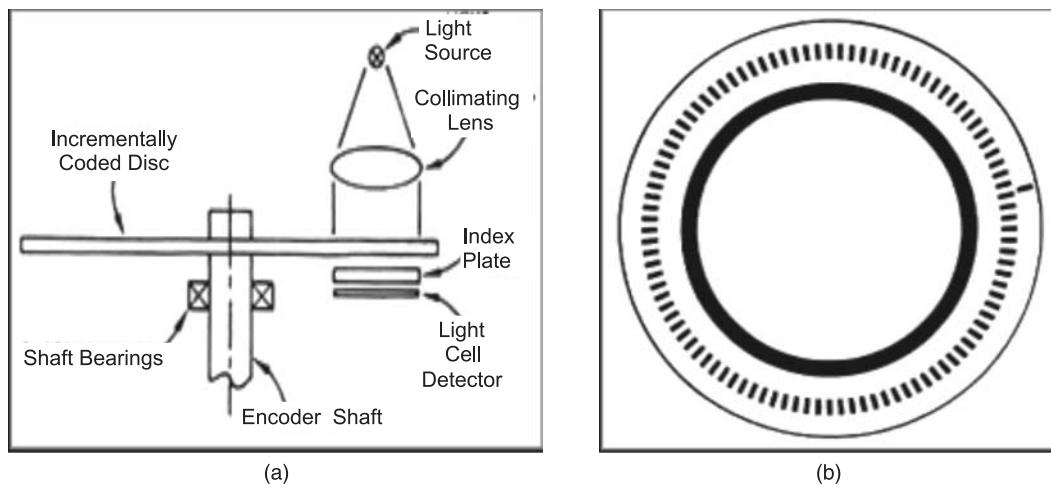


**Figure 13.34** Rotary optical encoder.

The incremental rotary encoder is an encoder with one set of pulses would not be useful since it could not indicate the direction of rotation. Most incremental encoders have a second set of pulses that is offset (out of phase) from the first set of pulses, and a single pulse that indicates each time the encoder wheel has made one complete revolution. Since the incremental encoder provides

only a string of pulses, a home switch must be used with this type of encoder to ensure that the encoder is calibrated to the actual location of the home reference point. It would be impossible to drill hundreds of holes in the encoder wheel to get the higher amounts of resolution because the wheel would not have enough material remaining to give the wheel strength. For this reason modern encoder wheels with high resolution use etched glass wheels. The glass is etched with chemicals to produce alternating opaque segments.

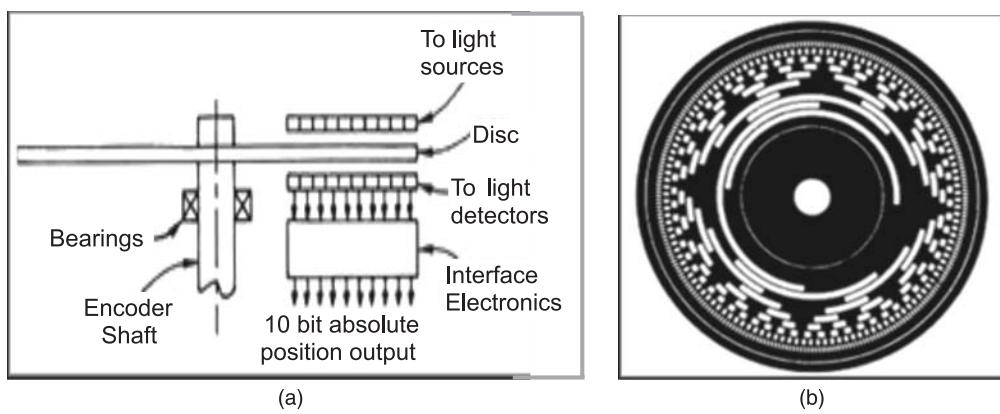
Figures 13.35(a) and (b) show an incremental encoder internal construction and the etched glass encoder. The glass encoder looks as if it has very thin black lines drawn on it. The black lines are the opaque segments that block light. The diagram from this figure shows only one light source and receiver. A second identical light source and receiver is mounted on the encoder in such a way that it produces the offset pulse train. One of the major drawbacks of the incremental encoder is that the number of pulses that are counted are stored in a buffer or external counter. If power loss occurs, the count will be lost. This means that if a machine with an encoder has its electricity turned off each night or for maintenance, the encoder will not know its exact position when power is restored. The encoder must use a home-detection switch to indicate the correct machine position. The incremental encoder uses a *homing routine* that forces the motor to move until a home limit switch is activated. When the home limit switch is activated, the buffer or counter is zeroed and the system knows where it is relative to fixed positional points.



**Figure 13.35** (a) Incremental encoder internal construction and (b) Etched glass incremental encoder wheel.

The absolute encoder is designed to correct this problem. It is designed in such a way that the machine will always know its location. Figure 13.36(a) shows absolute optical encoder internal construction and Figure 13.36(b) shows the pattern of concentric circles. This diagram also shows the location of 16 light sources and 16 light receivers that decode the pattern of light as it passes through the 16 concentric circle patterns. This type of encoder has alternating opaque and transparent segments like the incremental encoder, but the absolute encoder uses multiple groups of segments that form concentric circles on the encoder wheel like a “bull’s eye” on a target or dartboard. The concentric circles start in the middle of the encoder wheel and as the rings go out toward the

outside of the ring they each have double the number of segments than the previous inner ring. The first ring, which is the innermost ring, has one transparent and one opaque segment. The second ring out from the middle has two transparent and two opaque segments, and the third ring has four of each segment. If the encoder has 10 rings, its outermost ring will have 512 segments and if it has 16 rings, it will have 32,767 segments. Since each ring of the absolute encoder has double the number of segments of the prior ring, the values form numbers for a binary counting system. In this type of encoder there will be a light source and receiver for every ring on the encoder wheel. This means that the encoder with 10 rings has 10 sets of light sources and receivers, and the encoder with 16 rings has 16 light sources and receivers.



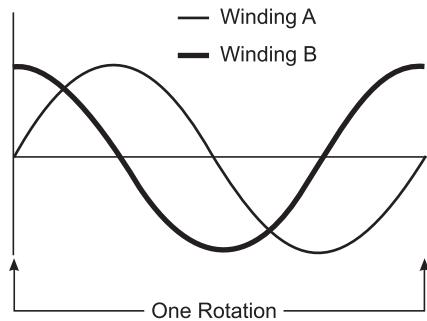
**Figure 13.36** (a) Absolute optical encoder internal construction and (b) Absolute encoder wheel.

The advantage of the absolute encoder is that it can be geared down so that the encoder wheel makes one revolution during the full length of machine travel. If the length of machine travel is 10 inches and its encoder has 16-bit resolution, the resolution of the machine will be 10/65,536, which is 0.00015 inch. If the travel for the machine is longer, such as 6 feet, a *coarse resolver* can keep track of each foot of travel, and a second resolver called the *fine resolver* can keep track of the position within 1 foot. This means the coarse encoder can be geared so that it makes one revolution over the entire 6-foot distance, while the fine encoder is geared so that its entire resolution is spread across 1 foot (12 inches). Since the absolute encoder produces only one distinct number or *bit pattern* for each position within its range, it knows where it is at every point between the two ends of its travel, and it does not need to be homed to the machine each time its power is turned off and on.

### 13.7.3 Resolvers

A resolver is an absolute position feedback device which operates as described below. The stator is made up of two windings—winding A and winding B. Winding A is positioned at a right angle to winding B. The rotor is made up of a third winding, winding C. This is energized with a sinusoidal voltage and allowed to rotate. The signal in winding C induces a signal in both windings A and B. Rotating winding C causes the magnitude of the induced signals to vary as a function of the angular position. The voltage induced in A is in quadrature to the voltage induced in B. Each

position along the rotation produces a different value for the combination of A and B. This is illustrated in Figure 13.37.



**Figure 13.37** Resolver.

Output from phase A is typically  $V_i * \sin(\omega t + \theta)$  and the output from phase B is typically  $V_i * \cos(\omega t + \theta)$ , where  $V_i$  = voltage in,  $\omega$  = Angular frequency,  $t$  = time and  $\theta$  = phase shift. Using the output of the two windings gives an absolute position, since each position has a different combination of A and B. The frequency also changes with the velocity, the velocity can also be determined. The data output from the two phases is usually converted from analog to digital by means of a resolver-to-digital converter. You can typically achieve a resolution up to 65,536 counts per revolution using resolvers.

### 13.7.4 Tachometers

Tachometers measure the angular speed of a rotating shaft. The speed of a shaft is measured in revolutions per minute (rpm). A tachometer could be as simple as a DC or AC generator that can determine the speed of shaft rotation by the amount of voltage the generator produces or the frequency of the output signal. The magnitude of the generator voltage and the frequency of the generated voltage will increase proportionally with speed. Frequency can also be measured by a photocell tachometer. The number of pulses produced by the photocell will increase as the speed of the shaft rotation increases.

Tachometers are used to determine the speed of a motor shaft for motor drives on conveyors, and to determine the speed of rotation of the screw shaft on a plastic injection molding machine. The speed of rotation of the screw on a plastic injection molding machine is important to control because the screw shaft is used to meter the amount of plastic that is drawn into the barrel of the machine for the next injection shot. If the speed is not controlled, the screw will turn at different speeds and more or less plastic will be drawn into the barrel and the amount of plastic being used for each part will be inconsistent.

The speed of the rollers in large rolling mills is also important to measure. In this type of application the rpm of each motor is measured and compared to setpoints. A servo system will increase or decrease the motor speed to keep the rollers at the correct rpm. In packaging applications several motors must be synchronized as the production line changes speeds to ensure the machine will operate correctly. Tachometers are used to measure the speed of each motor and a controller adjusts each motor speed to match the speed of the production line.

There are two basic types of tachometers that use the change in voltage to determine speed. These are the DC generator tachometer and the drag cup tachometer. These types of tachometers operate similarly to an unregulated generator. The faster they turn, the more voltage they produce. The DC generator produces a DC voltage, and the drag cup tachometer produces an AC voltage. These tachometers can provide the direction of rotation information as well as speed, which is useful because most of these types of sensors are used to provide feedback signals. Since these signals are basically a voltage, a simple voltmeter can be used as an indicator.

The *frequency-type tachometer* counts pulses produced by a rotating field tachometer, toothed rotor tachometer or photocell tachometer. These types of tachometers produce sine waves or pulses that can be counted. These types of tachometers need a more sophisticated digital circuit to complete the process of *count, store, calculate, display, and reset* to get a value displayed that represents rpm. The rotating field and the toothed rotor tachometers produce a waveform and the photocell uses a rotating disk that has a number of windows in it. A light source is positioned so that it will shine light through each window in the disk to a photocell detector as the disk spins. The disk is connected to the tachometer shaft, so when it turns, the windows line up with the photocell and the photocell produces a pulse when it is struck by light. In each of these types of tachometers, a pulse stream is produced and it is proportional to the speed of the tachometer shaft.

### 13.7.5 Magnetic Encoders

A magnetic encoder as in Figure 13.38 consists of a rotating gear made of ferrous metal and a magnetic pick-up that contains a permanent magnet and the sensing element. The gear, which is mounted on the rotating shaft, has precisely machined teeth that provide the code pattern. As the disk rotates, these teeth disturb the magnetic flux emitted by the permanent magnet, causing the flux field to expand and collapse. These changes in the field are sensed by the sensing element which generates a corresponding digital or pulse signal output.

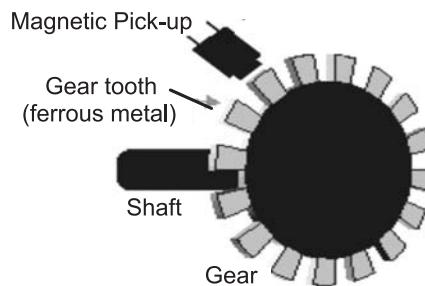


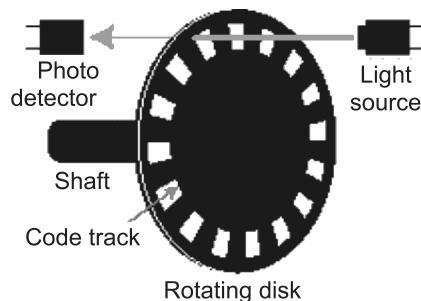
Figure 13.38 Magnetic encoders.

Two kinds of magnetic pick-ups exist:

1. **Hall effect**—These pick-ups use a semiconducting sensing element that relies on the Hall effect to generate a pulse for every gear tooth that passes the pick-up.
2. **Variable reluctance**—These pick-ups use a simple coil of wire in the magnetic field. As the gear teeth pass by the pick-up and disturb the flux, they cause a change in the reluctance of the gear/magnet system. This induces a voltage pulse in the sensing coil that is proportional to the rate flux change.

### 13.7.6 Optical Encoders

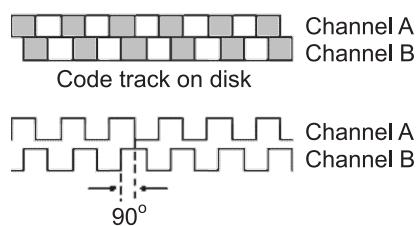
The most popular type of encoder is the optical encoder which consists of a rotating disk, a light source and a photodetector (light sensor) as shown in Figure 13.39. The disk, which is mounted on the rotating shaft, has coded patterns of opaque and transparent sectors. As the disk rotates, these patterns interrupt the light emitted onto the photodetector, generating a digital or pulse signal output. The encoding disk is made from glass, for high-resolution applications (11 to >16 bits) and plastic (Mylar) or metal, for applications requiring more rugged construction (resolution of 8 to 10 bits).



**Figure 13.39** Optical encoder.

### 13.7.7 Quadrature Encoders

The most common type of incremental encoder uses two output channels (A and B) to sense position as in Figure 13.40. Using two code tracks with sectors positioned 90° out of phase, the two output channels of the quadrature encoder indicate both position and direction of rotation. If A leads B, for example, the disk is rotating in a clockwise direction. If B leads A, then the disk is rotating in a counter-clockwise direction. By monitoring both the number of pulses and the relative phase of signals A and B, you can track both the position and direction of rotation. Some quadrature encoders also include a third output channel, called a *zero* or *index* or *reference signal*, which supplies a single pulse per revolution. This single pulse is used for precise determination of a reference position.

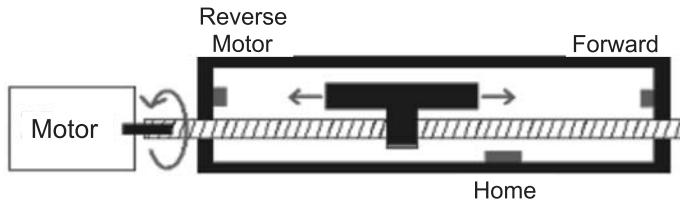


**Figure 13.40** Quadrature encoder.

## 13.8 MOTION I/O

Other I/O that is important in motion control includes limit switches, home switches, position triggers and position capture inputs. Limit switches provide information about the end of travel to help you avoid damaging your system. When a motion system hits a limit switch, it typically stops

moving. Home switches, on the other hand, indicate the system home position to help you define a reference point as in Figure 13.41. This is very important for applications such as pick-and-place.



**Figure 13.41** Limit and home switches in a motion control system.

Triggers such as position trigger outputs or position capture inputs help when integrating with other devices. With position trigger outputs (also called *breakpoints* and *position compare*), you can set up a trigger that executes at a prescribed position. This type of action is very useful in operations such as scanning, where you might want to trigger a system to take measurements at a series of prescribed positions. Position capture inputs, on the other hand, cause the motion controller to immediately capture an event occurrence position and store it in memory. This is useful if you have an external trigger and would like to know the position at which it occurs in your system.

### 13.8.1 Integration with Motion using RTSI

Many systems using motion control often do more than *just* motion control. Frequently, systems combine motion control with something else such as image acquisition or data acquisition. One of the challenges in integrating different processes is getting them to synchronize and work together. RTSI is one of the keys to getting motion control, data acquisition, and image acquisition to work in a coordinated fashion.

The RTSI (Real-Time System Integration) bus is a dedicated high-speed digital bus designed to facilitate systems integration by low-level, high-speed, real-time communication between National Instruments boards. Using the RTSI bus, motion boards can share high-speed digital signals with data acquisition, image acquisition, or digital I/O boards with no external cabling and without consuming bandwidth on the host bus. The RTSI bus also has built-in switching, so you can route signals to and from the bus on-the-fly through software.

For PCI boards, the physical bus interface is an internal 34-pin connector; signals are shared via a ribbon cable inside the PC enclosure. RTSI cables are available for chaining two, three, four, or five boards together. PXI modules require no cabling at all because the built-in PXI Trigger Bus handles RTSI functions. The RTSI connector has 34 pins but only seven are available for user signals. The software-configurable RTSI switch is used to accommodate more than seven signal options for each board. The switch is a digital many-to-few selector switch; any available signal can be routed to any RTSI pin. It is also possible to route more than one signal to a single RTSI pin or to connect two RTSI pins to the same signal.

RTSI provides high-speed, hardware-based synchronization capability to any automated measurement or machine control application, making it easy to:

- Build high-axis-count motion systems
- Clock data acquisition based on position
- Integrate your motion and vision together

---

## SUMMARY

---

- The components of a motion control system are application software, motion controller, amplifier or drive, motor, mechanical elements and feedback device or position sensor.
- Application software is divided into three main categories—configuration, prototype and application development environment (ADE).
- A motion controller acts as the brain of the motion control system and calculates each commanded move trajectory.
- The motor amplifier or drive is the part of the system that takes commands from the motion controller in the form of analog voltage signals with low current and converts them into signals with high current to drive the motor.
- The output stage of all servo amplifiers is an analog circuit.
- Most of the motors used in motion control are servo and stepper motors.
- Main differences between servo motors and stepper motors is that servo motors run using a control loop and require feedback of some kind.
- A control loop uses feedback from the motor to help the motor get to a desired state (position, velocity, and so on).
- Servomotors are available as AC or DC motors.
- Servomotors find application in press feed, in-line bottle filling, precision auger filling system, label applications and random timing infeed system.
- A stepper or stepping motor converts electronic pulses into proportionate mechanical movement.
- The most popular types of stepper motors are permanent-magnet (PM) and variable reluctance (VR) and hybrid stepper motors.
- The linear stepper motor is made flat instead of round and its motion is along a straight line instead of rotary.
- Stepper motors find applications in computer peripherals, business machines, process control and machine tool.
- Feedback devices help the motion controller know the motor location.
- Common feedback devices are absolute encoders, linear and rotary encoders, resolvers and tachometers.
- The most common position feedback device is the quadrature encoder which gives positions relative to the starting point.
- For relative position feedback, the most common device used is the incremental encoder.
- The most common absolute position feedback devices are the absolute encoders and resolvers.
- An absolute encoder is similar to an incremental encoder except that it does not need to have a reference position and can know what position the shaft is in on start up.
- A resolver is also another absolute position device that is commonly used for both position and velocity feedback.
- For velocity feedback, a tachometer is commonly used.
- RTSI stands (Real-Time System Integration) bus is a dedicated high-speed digital bus designed to facilitate systems integration.

---

**REVIEW QUESTIONS**

---

1. Draw and explain the different components of a motion control system.
2. Explain the three main categories of application software?
3. How does the motion controller acts as the brain of the motion control system and calculates each commanded move trajectory?
4. What is the basic function of a motor amplifier or drive? List the various types of drives used in industry.
5. List the advantages and disadvantages of servomotors compared to stepper motors.
6. Draw and explain the operation of a brushless servomotor.
7. Draw and explain the application of a servomotor in an in-line bottle filling plant.
8. Explain any one application of a stepper motor in an industry with a neat diagram.
9. Explain with neat diagrams the principle of operation and a typical industrial application of linear stepper motors.
10. List the wide variety of applications of stepper motors in industry.



# LabVIEW TOOL AND GSD APPLICATIONS

## 14.1 INTRODUCTION

The NI LabVIEW product family consists of the LabVIEW development environment and add-on software tools that extend LabVIEW graphical programming for specific applications. The add-on software toolkits are used for developing specialized applications and all the toolkits integrate seamlessly in LabVIEW. After you install a LabVIEW add-on such as a toolkit, module or driver, the documentation for that add-on appears in the LabVIEW Help or appears in a separate help system. You can access by selecting *Help»Add-On Help*, where *Add-On Help* is the name of the separate help system for the add-on.

The following list describes some of the signal processing and analysis tools; professional development tools to optimize, test, and distribute your VIs; third-party connectivity tools to Microsoft Office for professional reporting, databases to access and store data, and embedded design tools; and control design and simulation tools you have to extend the LabVIEW programming environment.

## 14.2 SIGNAL PROCESSING AND ANALYSIS

- **Digital Filter Design Toolkit:** It extends LabVIEW with functions and interactive design tools to rapidly explore classical designs and to design, model, and implement fixed-point and floating-point digital filters. You can design over 30 filter types—including FIR, IIR,

and multirate filters with well-known and special-purpose design options: Kaiser window, Dolph-Chebyshev, windowed, max flat, narrowband (interpolated FIR), elliptic, Chebyshev, Inverse Chebyshev, Butterworth, Bessel, notch/peak, max flat, comb, halfband multirate, single-stage multirate, n-stage multirate, Nyquist multirate, and root-raised/raised cosine multirate.

- **Sound and Vibration Toolkit:** It extends LabVIEW functions and indicators to handle audio measurements, fractional-octave analysis, swept sine analysis, sound-level measurements, frequency analysis, frequency response measurements, transient analysis, and several sound and vibration displays, including waterfall displays. Functionality includes scaling, calibration, limit testing, weighting, and distortion and single-tone measurements.
- **Modulation Toolkit:** It extends the built-in analysis capability of LabVIEW with functions and tools for signal generation, analysis, visualization, and processing of standard and custom digital and analog modulation formats. This toolkit provides quality measurements including EVM and modulation error ratio (MER); handles standard and custom modulation formats (AM, FM, PM, ASK, FSK, MSK, GMSK, PSK, QPSK, PAM, QAM, CPM); simulates and measures impairments including DC offset, IQ gain imbalance, and quadrature skew; and offers bit-error rate (BER), phase error, burst timing, and frequency deviation measurements.
- **Spectral Measurements Toolkit:** It extends LabVIEW with functions for acquiring and analyzing spectral measurements and performing modulation and demodulation on AM, FM and PM signals. This toolkit includes power spectrum, peak power and frequency, in-band power, adjacent-channel power, and occupied bandwidth, as well as 3D spectrogram capabilities.

### 14.3 PROFESSIONAL DEVELOPMENT TOOLS

- **LabVIEW Execution Trace Toolkit:** It provides real-time developers with an interactive tool for analyzing and verifying the execution of code they develop with the LabVIEW Real-Time Module. You can interactively analyze and benchmark thread and VI execution; optimize performance by identifying memory allocation, sleep spans, and resource contention; and create execution traces for LabVIEW Real-Time Module applications that you can print for documentation and code reviews.
- **Express VI Development Toolkit:** It provides tools to help you create interactive Express VIs that simplify the development of test, measurement and control applications. Express VIs provide an interactive, configuration-based, easy-to-use interface for your end users.
- **VI Analyzer Toolkit:** It pinpoints improvements you can make in the UI design, block diagram code, documentation, and VI properties and settings to optimize performance, usability, and maintainability of your VIs.

## 14.4 THIRD-PARTY CONNECTIVITY TOOLS

- **Report Generation Toolkit for Microsoft Office:** It provides a library of VIs for programmatically creating and editing Microsoft Word and Excel reports from LabVIEW.
- **Database Connectivity Toolkit:** It offers tools with which you can quickly connect to local and remote databases in which you store data and perform common database operations without having to perform structured query language (SQL) programming. This toolkit connects to the most popular databases through Microsoft ADO technology and readily connects to Microsoft Access, SQL Server, and Oracle databases.
- **Math Interface Toolkit:** It integrates LabVIEW VIs into The MathWorks, Inc. MATLAB® software environment, providing a better means of collaboration for development teams working in both LabVIEW and the MATLAB software. This toolkit converts LabVIEW VIs to native MATLAB MEX functions and allows you to easily distribute LabVIEW applications for native use in the MATLAB analysis environment. Using the Math Interface Toolkit, MATLAB users can take full advantage of LabVIEW's powerful I/O connectivity, graphical user interface, and over 2000 native functions directly from the MATLAB environment.
- **Simulation Interface Toolkit:** It gives design and test engineers a link between LabVIEW and The MathWorks, Inc. Simulink® and Real-Time Workshop® software to develop, prototype, and test dynamic systems using models developed in the Simulink simulation environment. You can create custom LabVIEW user interfaces using LabVIEW controls and indicators to interactively verify models you created in the Simulink environment. In addition, the LabVIEW Simulation Interface Toolkit provides a plug-in to Real-Time Workshop to import your models created in the Simulink environment into LabVIEW for deployment on real-time hardware platforms for hardware-in-the-loop (HIL) and other tests while making use of supported National Instruments FPGA, DAQ and CAN devices for real-time model I/O.

## 14.5 CONTROL DESIGN AND SIMULATION TOOLS

- **Control Design Toolkit:** It provides a library of VIs and LabVIEW MathScript functions that you use to design, analyze and deploy a controller for a linear time-invariant dynamic system model. This toolkit includes frequency response analysis tools such as Bode, Nyquist and Nichols plots; time response analysis tools such as step and impulse response analysis; classical design tools such as Root Locus; and state-feedback design tools such as Linear Quadratic Regulators and pole placement. In addition, the Control Design Toolkit supports PID design, lead-lag compensators, predictive and continuous observers, and recursive Kalman filters for stochastic system models that incorporate measurement and process noise. You can also use the Control Design Toolkit and the LabVIEW Real-Time Module to deploy a discrete controller to a real-time target.

- **PID Control Toolkit:** It offers PID and fuzzy logic control functions that you can combine with the math and logic functions already in LabVIEW to graphically develop control algorithms and programs for automated control.
- **Simulation Module:** It provides VIs, functions and other tools that you use to construct and simulate all or part of a dynamic system model. This module supports both nonlinear and linear dynamic system models and includes tools for trimming and linearizing nonlinear models. The LabVIEW Simulation Module includes functions for describing continuous and discrete transfer function, zero-pole-gain and state-space models, as well as nonlinear phenomena such as friction, deadband and backlash. You can interact with a model by using any of the VIs and functions included with LabVIEW itself. You also can use the Simulation Module and the LabVIEW Real-Time Module to deploy a continuous or discrete model to a real-time target.
- **Statechart Module:** It assists in large-scale application development by providing a framework in which you can build, debug and deploy statecharts in LabVIEW. With the LabVIEW Statechart Module, you can create a statechart that reflects a complex decision-making algorithm. Then, you can generate the block diagram code necessary to call the statechart from a VI. The Statechart Module supports hierarchy, concurrency and an event-based paradigm. If you install the appropriate LabVIEW module, you can execute statecharts on supported real-time targets and National Instruments FPGA devices.
- **System Identification Toolkit:** It combines data acquisition tools with system identification algorithms for accurate plant modeling. Use the LabVIEW System Identification Toolkit with National Instruments hardware, such as NI DAQ devices, to stimulate and acquire data from a plant and then identify a dynamic system model. This toolkit provides VIs that support parametric, nonparametric, partially-known, and recursive model estimation methods; AR, ARX, ARMAX, output-error, Box-Jenkins, transfer function, zero-pole-gain, and state-space model forms; and Bode, Nyquist, and pole-zero analysis. This toolkit also includes VIs for data preprocessing and model validation.

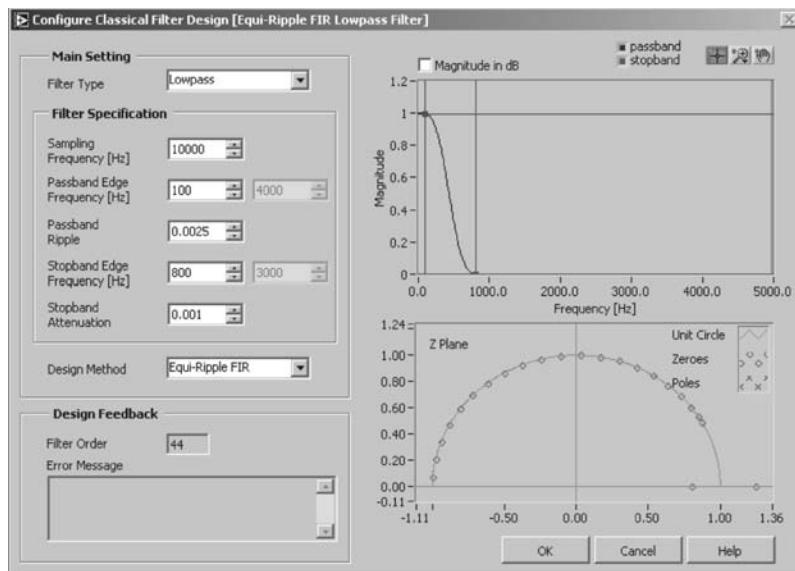
## 14.6 DIGITAL FILTER DESIGN TOOLKIT

The Digital Filter Design Toolkit provides a collection of digital filter design tools to supplement the LabVIEW Full Development System. The Digital Filter Design Toolkit helps you design digital filters without requiring you to have advanced knowledge of digital signal processing or digital filtering techniques. With the Digital Filter Design Toolkit, you can design, analyze and simulate floating-point and fixed-point digital filters. The features are as follows:

- Interactive and programmatic design, analysis and implementation of FIR/IIR digital filters within LabVIEW
- More than 30 filter types backed by more than 25 classical and modern design algorithms and 23 filter topologies
- Single-stage and multistage FIR filters for interpolation decimation and resampling

- Fixed-point filter design, analysis and simulation, including single-rate filters and multirate filters
- LabVIEW FPGA and ANSI C code generation for fixed-point filter implementation on an FPGA or DSP

Without prior knowledge about programming in LabVIEW, you can use the Digital Filter Design Express VIs to interact graphically with filter specifications to design appropriate digital filters. As an example Classical Filter Design Express VI is shown in Figure 14.1.



**Figure 14.1** Classical Filter Design Express VI.

The Digital Filter Design Toolkit provides VIs that you can use to design a digital finite impulse response (FIR) or infinite impulse response (IIR) filter, analyze the characteristics of the digital filter, change the implementation structure of the digital filter, and process data with the digital filter. In addition to the floating-point support, the Digital Filter Design Toolkit provides a set of VIs that you can use to create a fixed-point digital filter model, analyze the characteristics of the fixed-point digital filter, simulate the performance of the fixed-point digital filter, and generate fixed-point C code, integer LabVIEW code, or LabVIEW field-programmable gate array (FPGA) code for a specific fixed-point target.

The Digital Filter Design Toolkit includes VIs for floating-point multirate digital filter design. You can use the VIs to design a floating-point single-stage or multistage multirate filter, analyze the characteristics of the floating-point multirate filter, and process data with the floating-point multirate filter. In addition to the floating-point filter design, the toolkit also provides a set of VIs that you can use to create a fixed-point multirate filter, analyze the characteristics of the fixed-point multirate filter, simulate the behavior of the fixed-point multirate filter, and generate LabVIEW FPGA code from the fixed-point multirate filter for NI-RIO targets. In addition to the graphical tools for digital filter design, the toolkit also provides MathScript functions that LabVIEW MathScript supports. These MathScript functions enable you to design filters in a text-based

environment. You can perform signal processing and filtering on CompactRIO targets running the Wind River VxWorks real-time operating system (RTOS). You can specify the unit of measurement of the delay response in samples or seconds for the DFD Plot Group Delay VI and the DFD Plot Phase Delay VI. The online help includes tutorials on designing floating-point and fixed-point filters. You can access the tutorials by selecting Help»Search the LabVIEW Help from the pull-down menu in LabVIEW. On the Contents tab of the *LabVIEW Help*, choose *Toolkits»Digital Filter Design Toolkit»Digital Filter Design How-To*.

## 14.7 SOUND AND VIBRATION TOOLKIT

National Instruments' sound and vibration software provides a complete software solution for all audio, noise and vibration, and machine condition monitoring applications. Use the Sound and Vibration Toolkit to perform the following sound and vibration measurements:

- Scaling a signal to engineering units (EU)
- Calibrating a measurement channel
- Applying weighting filters
- Integrating time-domain signals
- Performing level measurements
- Performing swept-sine measurements
- Performing single-tone measurements
- Performing limit and mask testing
- Performing fractional-octave analysis
- Performing frequency analysis
- Performing transient analysis
- Performing distortion analysis
- Displaying results

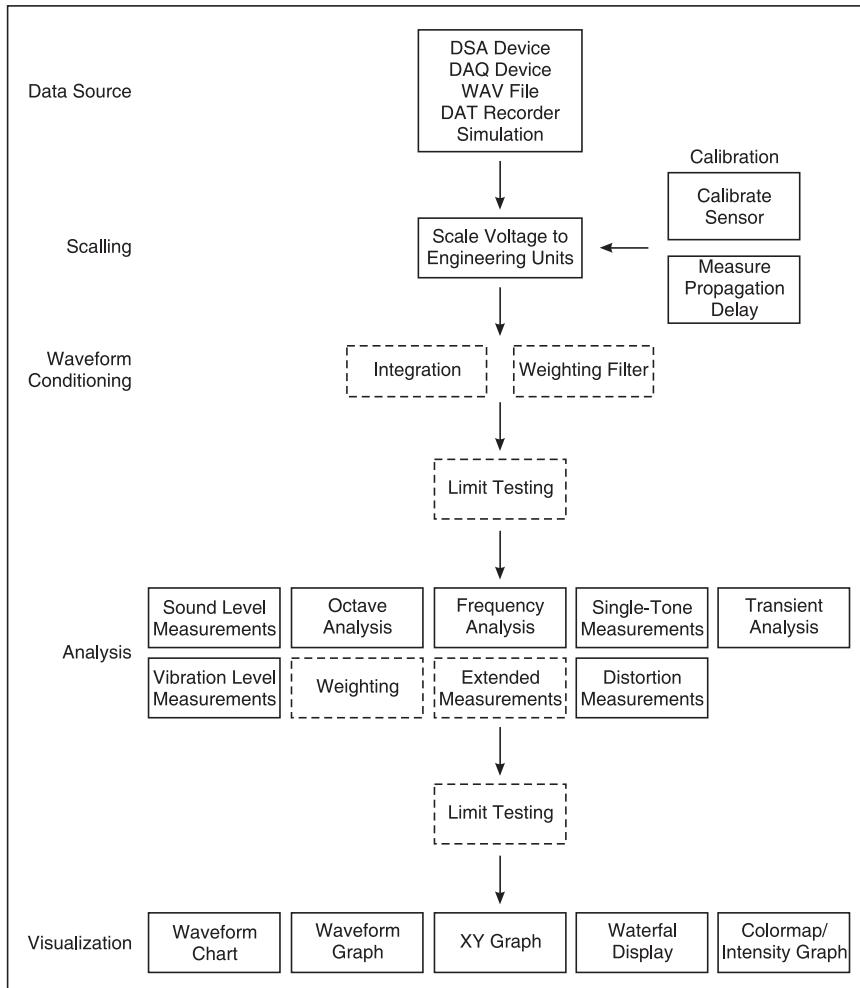
Based on an open-analysis capability and a flexible measurement library, the NI Sound and Vibration Measurement Suite and NI Sound and Vibration Toolkit present a unique software-based measurement approach to creating customized applications. The sound and vibration software packages consist of two components: NI Sound and Vibration Assistant and LabVIEW Analysis VIs. The Sound and Vibration Assistant is an interactive stand-alone software environment for quickly acquiring, analyzing, presenting, and logging acoustic and vibration measurements for all audio, noise and vibration, and machine condition monitoring applications. Rapid data acquisition hardware configuration and an interactive, drag-and-drop analysis library make it easy to quickly develop an application. The LabVIEW Analysis VIs provide additional LabVIEW functions for power spectra, frequency response functions (FRFs), fractional-octave analysis, sound level measurements, order spectra, order maps, order extraction, sensor calibration, human vibration filters, and torsional vibration. The Sound and Vibration Measurement Suite also offers more than 50 ready-to-run examples to simplify the process of getting started with your sound and vibration applications. The NI Sound and Vibration Measurement Suite is a collection of analysis and signal processing tools for noise, vibration, and harshness testing; machine condition monitoring; and audio test applications. Use the Sound and Vibration Measurement Suite to perform the following sound and vibration measurements:

- Scaling a signal to engineering units
- Calibrating a measurement channel
- Applying weighting filters
- Integrating time-domain signals
- Performing level measurements
- Performing fractional-octave analysis
- Performing frequency analysis
- Performing single-tone measurements
- Performing distortion analysis
- Performing swept-sine measurements
- Processing tachometer signals
- Performing order analysis
- Performing fault detection
- Measuring torsional vibration
- Performing transient analysis
- Performing limit and mask testing
- Displaying results
- Loading or saving signals to UFF58 (universal file format) files

You can use the Sound and Vibration Measurement Suite to perform measurements on acquired or simulated data. The Sound and Vibration Measurement Suite provides S&V Express Measurements Express VIs for you to develop and configure sound and vibration measurements interactively. You can use the Sound and Vibration Toolkit to perform measurements on digitized or simulated data. Figure 14.2 illustrates the sound and vibration measurement process. In Figure 14.2, the measurement operations shown on the Analysis line are not necessarily performed simultaneously. The dashed boxes in the figure indicate optional measurement operations.

## 14.8 MODULATION TOOLKIT

The NI Modulation Toolkit extends the built-in analysis capability of NI LabVIEW and LabWindows/CVI software with functions and tools for signal generation, analysis, visualization, and processing of standard and custom digital and analog modulation formats. With this toolkit, you can rapidly develop custom applications for research, design, characterization, validation, and test of communications systems and components that modulate or demodulate signals. The numerous Modulation Toolkit applications include digital modulation formats (AM, FM, PM, ASK, FSK, MSK, GMSK, PSK, QPSK, PAM, and QAM) that are the foundation of many digital communication standards found in 802.11a/b/g/n, ZigBee (802.15.4), WiMAX (802.16), RFID, satellite communications, and commercial broadcast among others. For RF applications, the Modulation Toolkit complements the NI PXI-5660 RF vector signal analyzer and the PXI-5671 RF vector signal generator. For lower-frequency operation (baseband or IF signals), the Modulation Toolkit works with the 100 MHz mixed-signal test platform with the digitizer, analog waveform generator, and digital waveform I/O products.



**Figure 14.2** Sound and vibration measurement process.

## 14.9 SPECTRAL MEASUREMENTS TOOLKIT

The National Instruments Spectral Measurements Toolkit provides a set of flexible spectral measurements in LabVIEW and LabWindows/CVI, including power spectrum, peak power and frequency, in-band power, adjacent-channel power and occupied bandwidth, as well as 3D spectrogram capabilities. In addition, the Spectral Measurements Toolkit contains VIs and functions for performing modulation-domain operations such as passband (IF) to baseband (I-Q) conversion, I-Q to IF conversion, and generation/analysis of analog modulated signals. The combination of these optimized algorithms and the GHz processing of your PC delivers unmatched measurement throughput. The Spectral Measurements Toolkit can be used with a variety of NI hardware including the PXI-5660 RF vector signal analyzer, digitizers and other modular hardware, as well as third-party

stand-alone instruments. The Spectral Measurements Toolkit contains LabVIEW VIs and LabWindows/CVI functions that perform the following operations:

- **Zoom frequency analysis:** Zoom fast Fourier transform (FFT) functions and VIs allow you to zoom in on a narrow frequency range in a spectrum.
- **Spectrum averaging:** The Spectral Measurements Toolkit supports averaging types such as root-mean-square (RMS) averaging, vector averaging and peak-hold averaging.
- **Spectral measurements:** The Spectral Measurements Toolkit contains functions and VIs that can measure power-in-band and adjacent channel power.
- **Unit conversion:** The Spectral Measurements Toolkit unit conversion supports typical radio frequency (RF) units, such as volts RMS squared ( $V^2_{rms}$ ), decibel (dB), decibel milliwatts (dBm), and dBm per hertz (dBm/Hz). You can use the Spectral Measurements Toolkit to convert a raw FFT spectrum to a power spectrum or power spectral density for noise measurements.
- **Peak power and frequency determinations:** The Spectral Measurements Toolkit includes a spectrum peak search algorithm that determines peak levels and frequency.
- **Zoom processing configuration:** The Spectral Measurements Toolkit configuration functions and VIs allow you to use conventional measurement settings such as center frequency, span and resolution bandwidth (RBW) to configure zoom processing. The configuration functions and VIs return an acquisition size based on your spectrum settings.
- **Spectrogram:** The Spectral Measurements Toolkit contains VIs that allow you to compute joint-time and frequency-domain calculations and display the results as a spectrogram. This feature is supported only in LabVIEW.
- **Analog modulation:** The Spectral Measurements Toolkit supports analog modulation to perform amplitude, frequency, and phase modulation and demodulation. Functions and VIs are included to perform upconversion and downconversion on baseband and passband signals.

You can use the Spectral Measurements Toolkit for the following applications:

- Frequency-domain measurements such as:
  - ◆ Adjacent channel power ratio (ACPR)
  - ◆ Channel spectrum
  - ◆ Power-in-band measurements
  - ◆ Average and peak power
  - ◆ Power spectral density
  - ◆ Spectrum limit and mask testing
- Modulation-domain measurements such as:
  - ◆ Frequency deviation
  - ◆ AM modulation index
- Component-level measurements such as characterization of oscillators, mixers and filters.

## 14.10 EXPRESS VI DEVELOPMENT TOOLKIT

The LabVIEW Express VI Development Toolkit allows you to create and edit Express VIs which you can distribute to users for building applications easily. This user guide contains five parts, each of which explains different aspects of how Express VIs work. Each part includes exercises that help you develop Express VIs. Throughout the exercises, you will learn how to use the Express VI Development Toolkit to create, edit and distribute Express VIs. An Express VI is a VI that allows users to interactively configure its settings through a dialog box. An Express VI consists of the following components:

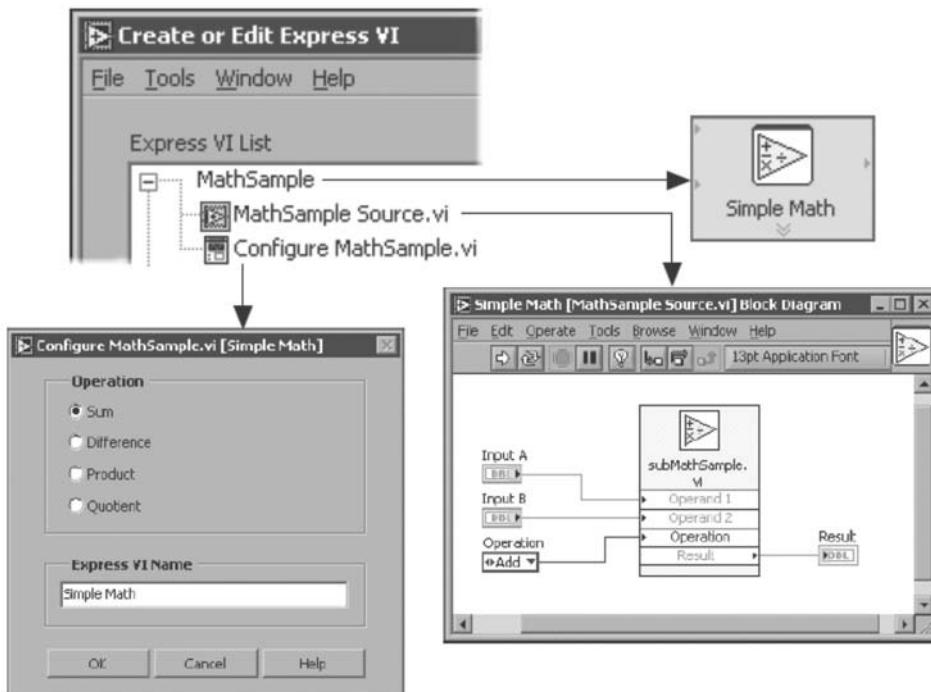
- **Configuration dialog box VI:** It allows users to configure settings for the run-time behavior of the Express VI.
- **Source VI:** It contains the code for the Express VI. The source VI also contains a link to the configuration dialog box VI.

The Express VI Development Toolkit saves both components in a library using the naming conventions as shown in Table 14.1. The Express VI Development Toolkit saves source VIs and configuration dialog box VIs in different LLBs that are stored together in the file system. If the source VI and configuration dialog box VI share subVIs, you can save the subVIs to a third LLB or place them in the most logical location. The source VI contains a link to its associated configuration dialog box VI. If you move the configuration dialog box VI to a different location relative to vi.lib or user.lib, the link breaks, and the Express VI prompts users to find the configuration dialog box VI. Figure 14.3 Graphical representation of an Express VI, using the MathSample Express VI as an example.

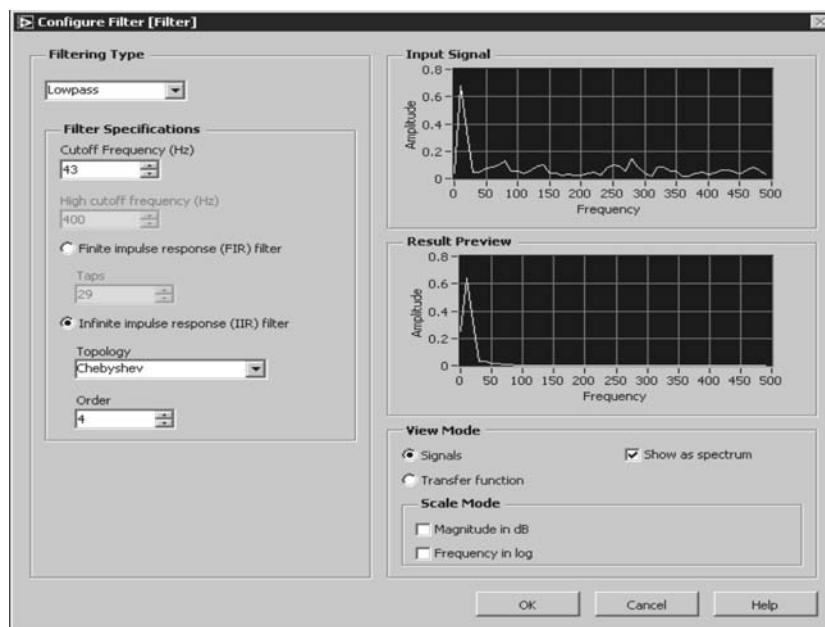
**TABLE 14.1** Express VI naming conventions

Component	Source VI Equivalent	Configuration Dialog Box VI Equivalent
<i>File Library</i>	<i>ExpressVINameSource.llb</i>	<i>ExpressVINameConfig.llb</i>
<i>VI Filename</i>	<i>ExpressVINameSource.vi</i>	Configure <i>ExpressVIName.vi</i>
<i>SubVI Filename</i>	<i>subExpressVIName.vi</i>	genHelp <i>ExpressVIName.vi</i>

Advantages of Using Express Vis are the primary benefit of Express VIs is their interactive configurability. Express VIs are useful when you want to give users a VI or library of VIs for building their own applications easily with minimal programming expertise. The configurability of Express VIs provides an interactive way to determine settings for operations that the user might not fully understand. For example, you might need a digital filter for a signal, but a library containing dozens of filter VIs does not help you choose the correct filter response or help you decide which parameters most directly affect the specific signal. However, the Filter Express VI allows you to interactively select the type of filter, manipulate filter parameters and view the filter response in different ways. For example, you can use the *Result Preview* graph in the configuration dialog box to examine the results of different filters and filter parameters directly on the signal as shown in Figure 14.4. After you run the Filter Express VI, the new signal displays as default data in the *Input Signal* and *Result Preview* graphs on the configuration dialog box.



**Figure 14.3** Graphical representation of an Express VI, using the MathSample Express VI.



**Figure 14.4** Using the Configuration dialog box to examine a noisy sine wave.

Another advantage of Express VIs is that separate instances of an Express VI function independently. For example, if you place a VI in five different areas on the same block diagram, the result is five exact copies of the VI. The source code, default values and front panel remain the same for all five copies. However, if you place an Express VI in five areas on a block diagram, the result is five separate Express VIs with different names, all independently configurable. Express VIs do not provide run-time interactive configuration for VIs. If you need run-time reconfiguration, build an application with a user interface that contains features similar to a configuration dialog box. Express VIs are maximized for ease of use. If you need an application to run with memory restrictions or high execution speeds, use standard VIs.

## 14.11 REPORT GENERATION TOOLKIT FOR MICROSOFT OFFICE

The LabVIEW Report Generation Toolkit for Microsoft Office provides VIs and functions you can use to create and edit reports in Microsoft Word and Excel formats from LabVIEW. In the LabVIEW Full and Professional Development Systems and the (LabVIEW 8.0 and later) Base Package, the Report Generation Toolkit modifies the VIs on the Report Generation palette to support Word and Excel formats in addition to HTML and standard LabVIEW report formats. You can use Report Generation Toolkit VIs to perform the following tasks:

- Create and edit Word documents and Excel worksheets.
- Use Word and Excel templates to create reports with a consistent, uniform style.
- Insert, format, and edit text, tables, images and graphs within Word documents and Excel worksheets.
- Run Microsoft Visual Basic for applications macros from Word documents and Excel workbooks.

When you create a report in LabVIEW, you must decide which type of report you want. Consider whether you need to print, save or email the report, and whether you want to use a template to create the report. You also should consider the applications that are available on other computers from which users access the report. You can use the Report Generation Toolkit to create the following types of reports:

- **Standard:** The standard LabVIEW report requires no additional software on the computer you use to create the report. You can insert text, tables and images into a standard report. You can print a standard report, but you cannot save the report. LabVIEW supports a limited number of formatting options, and you cannot use templates.
- **HTML:** HTML reports require no additional software on the computer you use to create the report. However, users need a Web browser to read the report. You can save an HTML report, but the report might not print with consistent formatting. LabVIEW supports a limited number of formatting options for HTML reports, and you cannot use templates.
- **Word and Excel:** Word and Excel reports require that users who want to read or print the reports have the appropriate application on their computers. You can insert text, tables, images and graphs into Word and Excel reports. You can save, print and email the reports. You can set a variety of formatting options with text and tables, and you can use templates to create a consistent report style.

You can set the report type in the New Report VI and the Report and MS Office Report Express VIs.

The Report and MS Office Report Express VIs provide easy ways to create reports in standard, HTML, Word and Excel formats. These Express VIs appear on the *Report Generation* palette with white backgrounds surrounded by a blue border. When you place an Express VI on the block diagram, a configuration dialog box appears that you can use to configure the settings for that instance of the Express VI.

## 14.12 SIMULATION INTERFACE TOOLKIT

The NI LabVIEW Simulation Interface Toolkit gives control system design and test engineers a link between the NI LabVIEW graphical development environment and The MathWorks, Inc. Simulink® software. With the LabVIEW Simulation Interface Toolkit, you can easily build custom LabVIEW user interfaces to view and control your simulation model during run time. This toolkit also provides a plug-in for The MathWorks, Inc. Real-Time Workshop® to import your models created in the Simulink environment into LabVIEW, so you can connect your model to the real world through a variety of real-time I/O platforms (requires the LabVIEW Real-Time Module). With these capabilities, you can easily take your models from software verification to real-world prototyping and hardware-in-the-loop simulation.

Using the LabVIEW Simulation Interface Toolkit, you can build custom user interfaces for models created in the Simulink environment. The SIT Connection Manager offers a configuration-based utility to connect a custom LabVIEW user interface to your models, eliminating the need for any programming knowledge. With the custom user interface, you can easily simulate, analyze, and verify your control model on a desktop PC. You can create a custom user interface for your simulation model in four steps:

**Step 1:** The LabVIEW Simulation Interface Toolkit adds a SignalProbe block to the Simulink environment. Place the SignalProbe in the top level of your model to enable LabVIEW to communicate with your model while it is running in the Simulink environment.

**Step 2:** Create a LabVIEW user interface by placing controls and indicators on a LabVIEW front panel.

**Step 3:** Use the SIT Connection Manager to specify the links between the LabVIEW user interface and your model.

**Step 4:** Start your simulation in the Simulink environment by selecting run in your LabVIEW user interface.

The LabVIEW Simulation Interface Toolkit also provides a plug-in for The MathWorks, Inc. Real-Time Workshop® software that helps facilitate the importation of your models into LabVIEW. When you use this capability with LabVIEW Real-Time, you can connect your model to the real world through a variety of real-time I/O platforms and still take advantage of the user interface capabilities used in the verification of the model. With the LabVIEW environment, you can select the appropriate real-time platform based on the performance, portability and durability requirements of your application.

The LabVIEW Simulation Interface Toolkit makes it easy to connect your models to a combination of analog, digital, and protocol-based I/O devices using the SIT Connection Manager. In addition to the user interface and I/O configuration, this utility helps you easily add stimulus to your application using multichannel data profiles. You also can specify multirate data logging on a per-channel basis to optimize file size and application performance. You can configure all of these capabilities without programming; however, you can further customize your configurations and add more features using LabVIEW graphical programming.

### **14.13 CONTROL DESIGN AND SIMULATION MODULE**

LabVIEW Control Design and Simulation Module can analyze open-loop model behavior, design closed-loop controllers, simulate online and offline systems and conduct physical implementations. The features are as follows:

- Construct plant and control models using transfer function, state-space, or zero-pole-gain.
- Analyze system performance with tools such as step response, pole-zero maps and Bode plots.
- Simulate linear, nonlinear and discrete systems with a wide option of solvers.
- Deploy dynamic systems to real-time hardware using built-in functions and LabVIEW Real-Time Module.

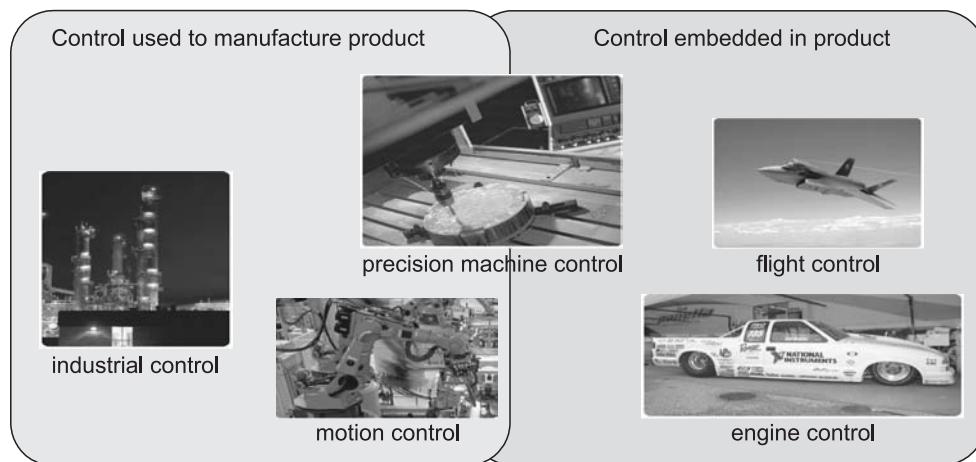
Create models from first principles using transfer function, state-space or zero-pole-gain representation. With time and frequency analysis tools, such as time step response or bode plot, you can interactively analyze open- and closed-loop behavior. Use built-in tools for both multiple input, multiple output (MIMO) and single input, single output (SISO) systems and take advantage of simulation capabilities to verify linear and nonlinear system dynamics. You can also use built-in tools to convert your models developed in The MathWorks, Inc. Simulink® software to work with LabVIEW.

You also can expand LabVIEW Control Design and Simulation usability with other NI software tools. For example, you can use the LabVIEW System Identification Toolkit to find empirical models from real plant stimulus-response information and the LabVIEW Statechart Module for event-based control design or event-driven simulation. You can deploy dynamic systems to real-time hardware targets with the LabVIEW Real-Time Module for rapid control prototyping and hardware-in-the-loop applications.

### **14.14 CONTROL DESIGN TOOLKIT**

LabVIEW has a PID control toolkit that can be used to solve these applications. However, while PID control is sometimes sufficient for a given control application, there are still several cases where this doesn't make sense. In some cases, a more complex higher-order controller is required or the controller must compensate for non-linear behavior in the plant system. Sometimes the PID control algorithm is incapable of achieving the required performance. Finally, because the PID control algorithm is a single-input single-output controller, it cannot be used to handle complex multi-input multi-output systems. For instance, to balance and control the position of a helicopter, you need to read in several inputs and control several outputs.

Control systems are commonly found in industrial environments as shown in Figure 14.5. For example, consider an oil refinery with process control systems that continually manufacture and produce oil. The control system used for processing may consist of a Programmable Logic Controller (PLC) executing a PID algorithm, or a Distribute Control System (DCS) for a larger process control. In this case, the control system is used to manufacture a product. A control system can also be part of an end-product being manufactured. This has been seen primarily in the automotive and aerospace industries with electronic control units and flight control systems. However, control systems are now finding their way into other end products such as precision motor controllers for computer hard drives and white goods like washing machines.



**Figure 14.5** Applications of control systems.

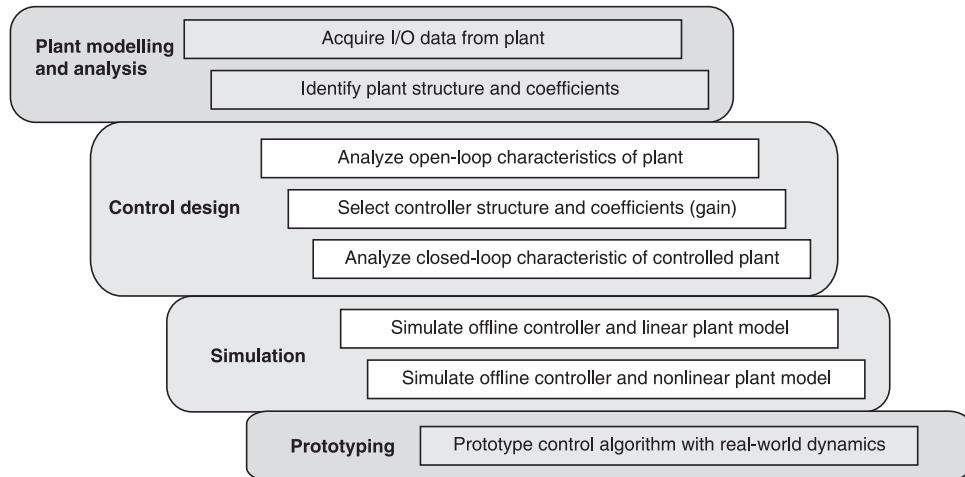
While control systems used to manufacture a product often stem from established control strategies such as PID control, control systems embedded in end-products often use new and innovative control strategies. The tools and techniques used to develop and embed control systems in end-products has evolved to include model-based design tools. However, manufacturing control engineers are also beginning to adopt these tools and techniques to develop more advanced control systems.

In each of these cases, there is a need for a custom control algorithm. When designing these control systems, control engineers use a model-based design approach. This consists of using mathematical models to represent plants and controllers. Model-based control design can be broken down into four steps as shown in Figure 14.6. PID is commonly used because many people understand it and it is good enough. The control design tools can help tune the gains.

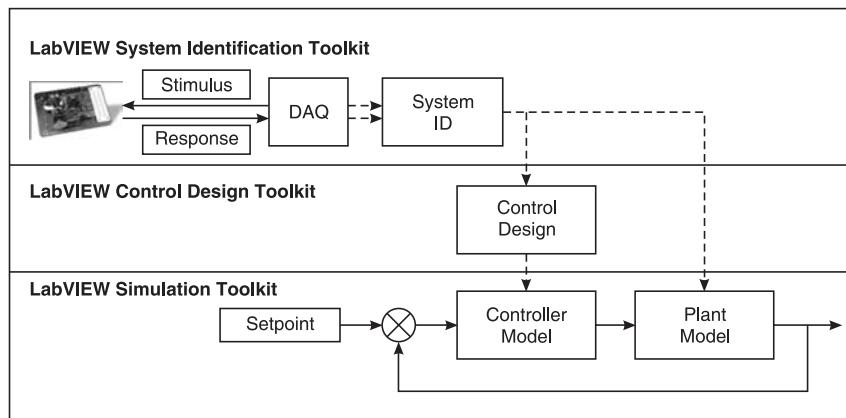
Figure 14.7 shows the LabVIEW Control Design Tools. Using the LabVIEW System Identification Toolkit, a design engineer will begin the control design process by creating a model of a plant. This begins with acquiring stimulus response data from the plant using standard data acquisition hardware and then selecting an plant structure that fits this data. From there, the engineer will find the right coefficients for the plant model that accurately represent the plant.

Next, the engineer will use the LabVIEW Control Design Toolkit to develop a controller for the plant. First, the engineer will analyze the open loop characteristics of the plant and then select a basic control structure to control this plant. From there, he/she will determine the right coefficients

for this controller. Finally, the engineer will connect the controller with the plant and analyze the close-loop characteristics of the controlled plant.



**Figure 14.6** Model-based control design process.



**Figure 14.7** LabVIEW control design tools.

Finally, the LabVIEW Simulation Toolkit enables the engineer to simulate the controller with a linear plant model. Next, the engineer can extend the simulation to a non-linear plant model to accommodate for non-linear operating conditions in the real-world. Finally, the engineer can take the model and run it directly on real-time hardware to prototype the control system and test it against real-world dynamics. Figure 14.9 LabVIEW Control Design Tools.

## 14.15 PID CONTROL TOOLKIT

Currently, the Proportional-Integral-Derivative (PID) algorithm is the most common control algorithm used in industry. Often, people use PID to control processes that include heating and

cooling systems, fluid level monitoring, flow control and pressure control. In PID control, you must specify a process variable and a setpoint. The process variable is the system parameter you want to control, such as temperature, pressure, or flow rate, and the setpoint is the desired value for the parameter you are controlling. A PID controller determines a controller output value such as the heater power or valve position. The controller applies the controller output value to the system which in turn drives the process variable toward the setpoint value. You can use the PID VIs with National Instruments hardware to develop LabVIEW control applications. Use I/O hardware, like a DAQ device, FieldPoint I/O modules, or a GPIB board, to connect your PC to the system you want to control. You can use the I/O VIs provided in LabVIEW with the LabVIEW PID Control Toolkit to develop a control application or modify the examples provided with the toolkit.

Use the PID VIs to develop the following control applications based on PID controllers:

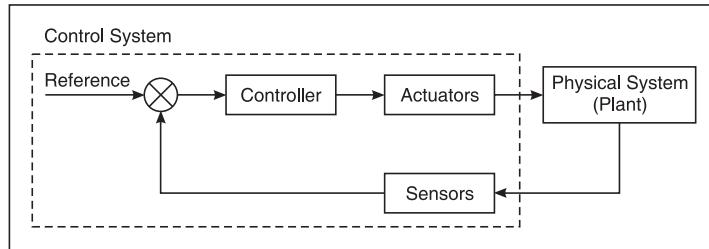
- Proportional (P), proportional-integral (PI), proportional-derivative (PD) and proportional-integral-derivative (PID) algorithms
- Gain-scheduled PID
- PID autotuning
- Error-squared PID
- Lead-lag compensation
- Setpoint profile generation
- Multi-loop cascade control
- Feedforward control
- Override (minimum/maximum selector) control
- Ratio/bias control

You can combine these PID VIs with LabVIEW math and logic functions to create block diagrams for real control strategies. The PID VIs use LabVIEW functions and library subVIs, without any Code Interface Nodes (CINs), to implement the algorithms. You can modify the VIs for your applications in LabVIEW, without writing any text-based code.

#### 14.16 SIMULATION MODULE

Simulation is a process that involves using software to recreate and analyze the behavior of dynamic systems. You use the simulation process to lower product development costs by accelerating product development. You also use the simulation process to provide insight into the behavior of dynamic systems you cannot replicate conveniently in the laboratory. For example, simulating a jet engine saves time, labor, and money compared to building, testing, and rebuilding an actual jet engine. Figure 14.8 shows a sample dynamic system.

The dynamic system in Figure 14.8 represents a closed-loop system also known as a *feedback system*. In closed-loop systems, the controller monitors the output of the plant and adjusts the actuators to achieve a specified response. You can use the LabVIEW Simulation Module to simulate a dynamic system or a component of a dynamic system. For example, you can simulate only the plant while using hardware for the controller, actuators and sensors. This chapter provides an overview of the simulation process and describes how to use the Simulation Module to simulate a dynamic system.



**Figure 14.8** Sample dynamic system.

### 14.17 SYSTEM IDENTIFICATION TOOLKIT

The NI LabVIEW System Identification Toolkit combines data acquisition tools with system identification algorithms for accurate plant modeling. You can take advantage of LabVIEW intuitive data acquisition tools such as the DAQ Assistant to stimulate and acquire data from the plant and then automatically identify a dynamic system model. You can convert system identification models to state-space, transfer function, or pole-zero-gain form for control system analysis and design. The toolkit includes built-in functions for common tasks such as data preprocessing, model creation and system analysis. Using other built-in utilities, you can plot the model with intuitive graphical representation as well as store the model.

You can purchase the toolkit separately or as part of NI Developer Suite, a modular product offering with which you can select software components based on your application needs. With an NI Developer Suite subscription, you receive a new set of CDs featuring the most recent software version for each product in the suite four times a year as well as direct access to technical support from NI applications engineers via phone and e-mail.

### 14.18 DIAdem

It is easy working with DIAdem for data mining, analysis and report generation. DIAdem is the interactive National Instruments software for finding and managing technical data, mathematically and graphically analyzing the data and presenting the data in reports. You search for data on your computer drives or on the network, and navigate in data files and databases to load the data you find, into DIAdem. You view the loaded data to decide which data to run mathematical analyses on. You present calculation which results in a report. If you frequently use the same method to evaluate data, you create a script that automates evaluations.

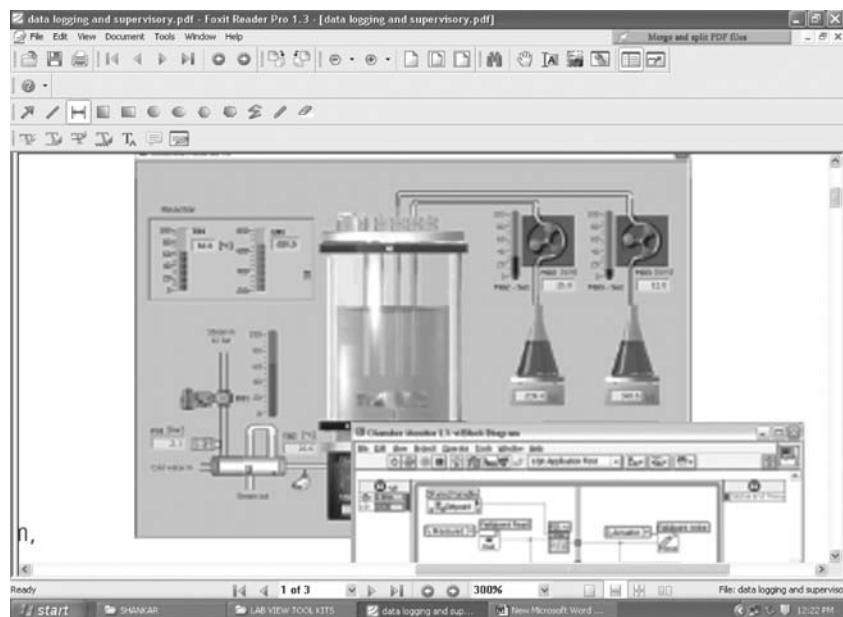
DIAdem consists of several panels. You use the panel bar, which is always available on the left edge of the DIAdem screen, to switch panels. Each panel deals with a particular type of task. You use DIAdem NAVIGATOR to mine and load data in different file formats. You use DIAdem VIEW to view data, to graphically analyze curve sections and to edit data, for example, to correct errors. You use DIAdem ANALYSIS to evaluate data mathematically with standard functions or your own formulas. You use DIAdem REPORT to create multi-page reports that document data and present results. DIAdem SCRIPT combines all the panel functions in scripts. You can use scripts to create your own applications that process tasks automatically.

When you switch DIAdem panels, the user interface changes for quick access to the functions you want. Each DIAdem panel has its own group bar to the right of the panel bar. Click a button on the group bar and select a function from the function group that opens. The workspace also changes with the DIAdem panel and displays a folder and file overview or a worksheet, for example. Each panel has its own toolbar, menus, and shortcut menus which contain frequently-used functions.

#### 14.19 DATA LOGGING AND SUPERVISORY CONTROL

The National Instruments LabVIEW Datalogging and Supervisory Control (DSC) Module is the best way to interactively develop your high-channel-count and distributed monitoring and control systems. The NI LabVIEW DSC Module extends the LabVIEW development environment to interactively configure and manage alarms and events, efficiently log data to a distributed historical database, view real-time and historical data, set application security, and easily network LabVIEW Real-Time targets and other OPC devices to create one complete system. Whether you need to build a full-scale industrial automation and control system, configure thousands of channels in a data-logging application, or just monitor a few I/O points for historical collection, the LabVIEW DSC Module provides the tools to make you more productive.

Figure 14.9 shows the front panel of DSC Module. The LabVIEW DSC Module provides built-in utilities for data logging and alarm management, as well as real-time and historical trending. Whether you are collecting data from National Instruments dataacquisition products, LabVIEW Real-Time targets, Compact FieldPoint or CompactRIO modules, or programmable logic controllers, you can quickly configure the I/O you want and use the LabVIEW DSC Module to automatically acquire the data. With the LabVIEW DSC Module, you can automatically monitor and log alarms



**Figure 14.9** Front panel of Datalogging and Supervisory Control (DSC) Module.

and events for your system. The LabVIEW DSC Module makes building distributed applications easy and intuitive. Simply browse the network for the I/O you want to access around your lab, your production floor, or the world. Using LabVIEW DSC built-in security, you determine which machines have read-only access, read-and-write access, or any access at all. The LabVIEW DSC Module adds full OPC client and server capabilities to your LabVIEW applications as well, so you can communicate with any OPC server available on the market today. These servers manage device I/O and communication status information. With the LabVIEW DSC Module, security is built into the LabVIEW environment and implemented across the network seamlessly. You can add system-level and operator interface security to any existing or new LabVIEW application with no programming.

## 14.20 EMBEDDED MODULE

The LabVIEW Embedded Module facilitates dataflow graphical programming for embedded systems, and includes hundreds of analysis and signal processing functions, integrated I/O and an interactive debugging interface. The complete ADI VisualDSP++ development and debugging environment is included with the LabVIEW Embedded Module for Blackfin Processors, so you can easily deploy LabVIEW code along with fully integrated debugging capabilities.

By seamlessly integrating with the VisualDSP++ compiler, linker, and debugging interface, you have the ability to easily single step through your graphical code in LabVIEW while visualizing the embedded code—both C and ASM – within the debugger. With this module, you can integrate VisualDSP++-specific compiler options into LabVIEW such as the ability to enable cache, optimize linking, and view live front-panel updates via background telemetry channel (BTC) technology on-chip JTAG debugging.

The module includes a standardized component device driver library for Blackfin evaluation hardware onboard peripherals. A common component driver model provides consistent representation and a driver you can reuse from design to design. The component driver architecture supports multiple devices running simultaneously in a system. The LabVIEW Embedded Module for Blackfin Processors includes the VisualDSP++ Kernel (VDK), providing state-of-the-art scheduling and resource allocation tailored specifically to address the memory and timing constraints of high-performance applications. The module takes advantage of VDK to work with multiple threads, time processes and OS services. By combining the dataflow graphical programming power with the LabVIEW Embedded Module for Blackfin Processors, you gain real-time, front-panel debugging capabilities.

## 14.21 BIOMEDICAL STARTUP KIT

The Biomedical Startup Kit is an add-on palette to LabVIEW designed for those interested in the fields of life science and biomedical engineering. It offers an easy-to-use API for programming some of the more common tasks in LabVIEW, as well as functions commonly used in biomedical science laboratories. The palette as shown in Figure 14.10 includes sub VIs for acquiring and logging data, controlling laboratory instruments, processing biomedical images and signals, and using text-based programming tools commonly employed in processing biomedical data. Of particular interest to

some users is the Simulate ECG Signal Express VI bundled in the palette as shown in Figure 14.11. A number of features of the Biomedical Startup Kit palette lend themselves to getting users of LabVIEW up and running quickly. One novel feature of the palette is the Shell VI, which allows dropping an entire DAQmx programming model onto the block diagram with a single click.

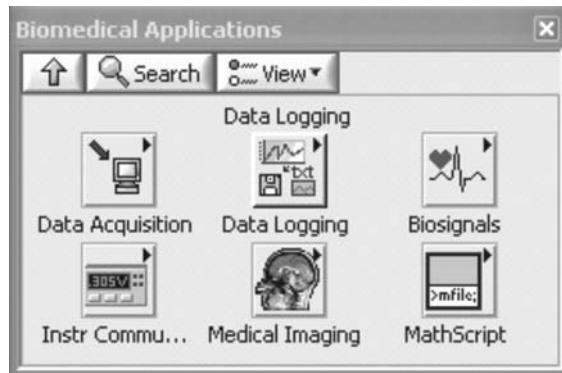


Figure 14.10 Biomedical Startup Kit.

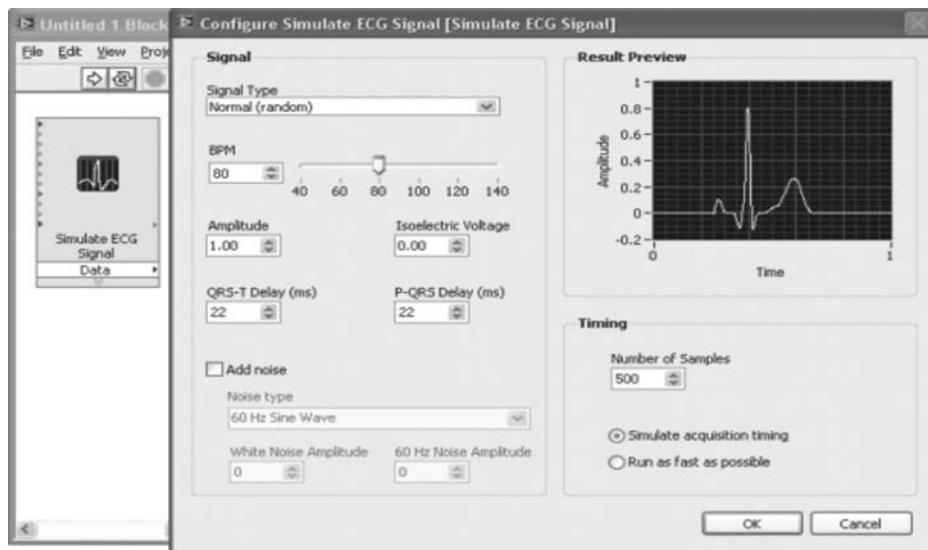


Figure 14.11 Simulate ECG Signal Express VI.

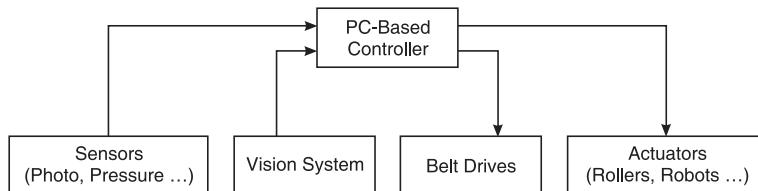
## 14.22 GSD APPLICATIONS

### 14.22.1 Material Handling System

Typical systems in material handling include a vision system, belt drives as well as system specific sensors and actuators depending on system needs as in Figure 14.12. Sensors include photo-sensors, pressure transducers, pneumatic-controlled rollers as well as handling robots. The vision system is

chiefly responsible for unit identification and inspection but can be used to determine other information such as alignment. The belt drives are critical since they are responsible for the primary movement and transport of units in the distribution center including motion on the main belt itself. Sensors can be used to determine position along the belt and other information including size and weight. There are also invariably other actuators and pneumatics used to control sorting, picking and placing of units. Software package requirements and description are given in Table 14.2. Table 14.3 shows the various functions, its signal and solutions required for PCI based Configuration. Similarly, Table 14.4 provides the requirements when it is PXI based configuration. Table 14.5 lists the Compact FieldPoint-based configuration. The advantages are as follows:

- Tight integration between data acquisition, real-time controllers, vision system and motion controllers
- Shorter time to deployment with LabVIEW
- Scalable to future requirements
- Easily integrates with existing control systems and software



**Figure 14.12** Material handling system.

**TABLE 14.2** Software requirements for material handling system

Package	Description
NI Developer Suite LabVIEW Professional Control Edition	With LabVIEW, you can rapidly create automation applications using intuitive graphical development. LabVIEW integrates measurements, vision and motion into one platform. The Professional Control Edition includes the LabVIEW Real-Time and LabVIEW Data Logging and Supervisory Control modules.
LabVIEW Real-Time	LabVIEW Real-Time and RT Series hardware deliver deterministic, real-time control needed for machine monitoring and test systems. You can choose LabVIEW RT Series hardware targets based on PXI and FieldPoint depending on the performance and I/O requirements of your industrial machine monitoring and control system.
LabVIEW Data Logging and Supervisory Control	LabVIEW Data Logging and Supervisory Control offers data management tools, such as easy-to-use I/O configuration for large-channel-count applications, automatic data logging, full alarm management and event logging, and real-time and historical trending. With easy networking, including a networked database for distributed logging, built-in security, and OLE for process control (OPC) connectivity, the LabVIEW Data Logging and Supervisory Control Module provides tremendous ease of use to get your high-channel-count system up and running quickly

**TABLE 14.3** PCI-based configuration

<i>Function</i>	<i>Signal</i>	<i>Solution</i>
Vision Monitoring System	Image	Image Acquisition Board
Belt Drives	N/A	Motion Controller / Motor Drive
Photo-Sensors	Digital	Digital Input Board
Pressure-Transducer	Analog	DAQ board w/ Signal Conditioning

**TABLE 14.4** PXI-based configuration

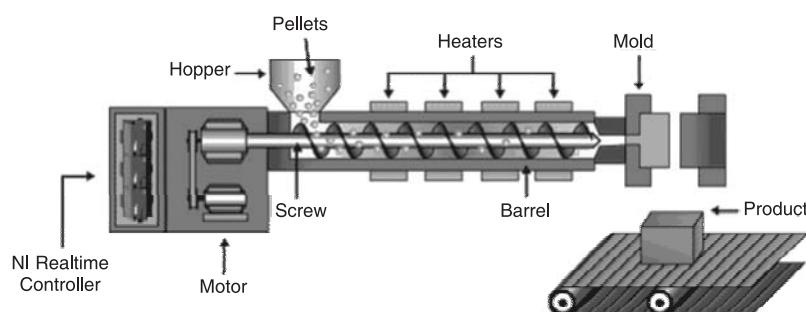
<i>Function</i>	<i>Signal</i>	<i>Solution</i>
Vision Monitoring System	Image	Image Acquisition Board
Belt Drives	N/A	Motion Controller/Motor Drive
Photo-Sensors	Digital	Digital Input Board (need signal conditioning)
Pressure-Transducer	Analog	DAQ board w/Signal Conditioning

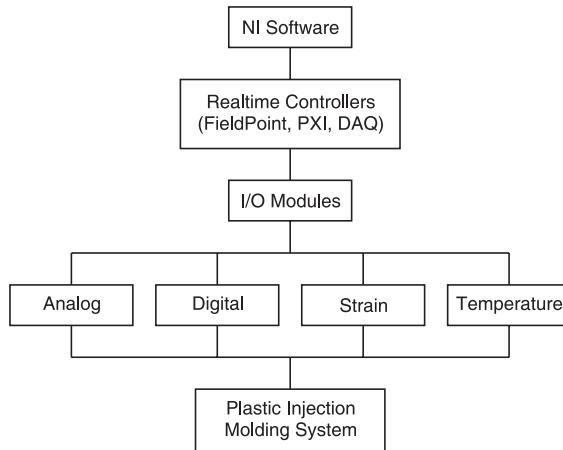
**TABLE 14.5** Compact FieldPoint-based configuration

<i>Function</i>	<i>Signal</i>	<i>Solution</i>
Photo-Sensors	Digital	Digital Input Module
Pressure-Transducer	Analog	Analog Input Module

#### 14.22.2 Plastic Injection Molding System

Manufacturers use thermoplastics to make a large array of products in industries from automotive and telecom to food packaging and toys. In this manufacturing process, pellets of thermoplastic material are mixed and carefully heated under PID control. The melted plastic slurry is then injected into a mold and cooled to form the plastic product. Careful monitoring and control are required to obtain high yields. Figure 14.13 is a representation of the typical plastic injection molding system. The process starts as the embedded controller mixes the thermoplastic pellets and then empties the beads into the heating barrel. After the plastic pellets have uniformly reached their flow temperature, pressure is built in the heating shaft and the melted plastic is injected into the mold. Figure 14.14 shows the block diagram of plastic injection molding system using NI products.

**Figure 14.13** Typical plastic injection molding system.



**Figure 14.14** Block diagram of plastic injection molding system.

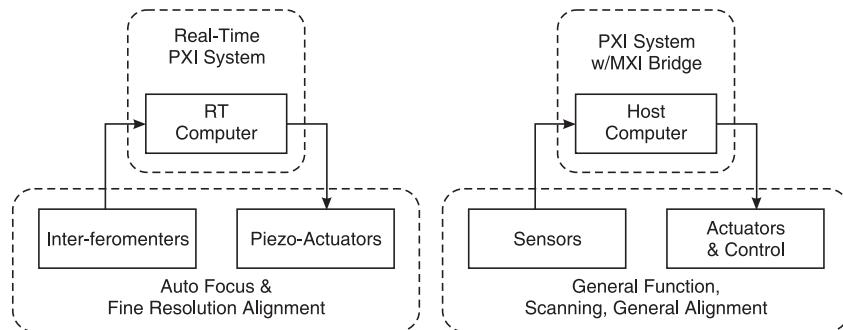
The software requirements are as follows:

- NI Developer Suite Professional Control Edition with LabVIEW enables you to rapidly create automation applications using intuitive graphical development. LabVIEW integrates measurements, vision and motion into one platform. The Professional Control Edition includes the LabVIEW Real-Time and LabVIEW Data Logging and Supervisory Control modules.
- LabVIEW Real-Time and RT Series hardware deliver deterministic, real-time control needed for machine monitoring and test systems. You can choose LabVIEW RT Series hardware targets based on PXI and FieldPoint depending on the performance and I/O requirements of your industrial machine monitoring and control system.
- LabVIEW Data Logging and Supervisory Control offers data management tools such as easy-to-use I/O configuration for large-channel-count applications, automatic data logging, full alarm management and event logging, and real-time and historical trending. With easy networking, including a networked database for distributed logging, built-in security, and OLE for process control (OPC) connectivity, the LabVIEW Data Logging and Supervisory Control Module provides tremendous ease of use to get your high-channel-count system up and running quickly.

#### 14.22.3 Semiconductor Production Control System

Holographic lithography is quickly emerging as the preferred method for patterning semiconductor substrates. One of the more difficult challenges in semiconductor production and the lithography process is the alignment of the mask and the wafer. Typically, wafers must be maintained at a fixed position relative to the lithography mask by means of actuators. The variation in position must be less than 0.2 μm for a successful etch. For successful automation of the semiconductor production process, this step must be carefully controlled. National Instruments offers a wide variety of tightly integrated DAQ, motion and real-time controllers to effectively solve the automation of the semiconductor lithography process. In a typical lithography process, there could be six or more systems or stages as shown in Figure 14.15. These system or stages consist of:

1. Loading substrate
2. Pre-alignment
3. Auto-focus & Auto-alignment
4. Scanning stages
5. Microscope alignment
6. Unloading substrate



**Figure 14.15** Semiconductor production control system.

Loading the substrate consists of placing the substrate on an air cushion. Once on the air cushion, two retractable reference pins are used for pre-alignment to align the wafer on a third pin at which control of the wafer switches from the air cushion to a vacuum which clamps the wafer to a chuck. The table has two axes for the X/Y direction and 3 axes for the vertical movement. For high-precision auto-alignment of the wafer and chuck piezo-actuators are used and controlled via a real-time control system to position and maintain this position between the wafer and the auto-focusing system. Several motorized stages will be used to perform raster scanning to uniformly expose the plate, one for the UV beam and one for the focus beam. Both beams are tightly synchronized in hardware. To perform the final alignment, microscopes are used in conjunction with a vision system to ensure that alignment marks in the hologram and on the substrate align.

## SUMMARY

- The NI LabVIEW product family consists of the LabVIEW development environment and add-on software tools that extend LabVIEW graphical programming for specific applications.
- NI LabVIEW Toolkits are grouped as signal processing and analysis tools, professional development tools, third-party connectivity tools and control design and simulation tools.
- Signal Processing tools include Digital Filter Design Toolkit, Sound and Vibration Toolkit, Modulation Toolkit and Spectral Measurements Toolkit.
- Professional Development tools include LabVIEW Execution Trace Toolkit, Express VI Development Toolkit and VI Analyzer Toolkit.
- Third-Party Connectivity Tools include Report Generation Toolkit for Microsoft, Database Connectivity Toolkit, Math Interface Toolkit and Simulation Interface Toolkit.
- The LabVIEW Express VI Development Toolkit allows you to create and edit Express VIs.
- Control Design and Simulation tools include Control Design Toolkit, PID Control Toolkit, Simulation Module, Statechart Module, System Identification Toolkit.

- System Identification Toolkit helps begin the control design process by creating a model of a plant, with acquiring stimulus response data from the plant using standard data acquisition hardware and then selecting an plant structure that fits this data.
- Control Design Toolkit helps develop a controller for the plant, first to analyze the open loop characteristics of the plant and then select a basic control structure to control this plant. We can determine the right coefficients for this controller and can connect the controller with the plant and analyze the close-loop characteristics of the controlled plant.
- LabVIEW Simulation Toolkit enables to simulate the controller with a linear plant model. It extends the simulation to a non-linear plant model to accommodate for non-linear operating conditions in the real-world and can take the model to run on real-time hardware to prototype the control system and test it against real-world dynamics.
- LabVIEW Data Logging and Supervisory Control offers data management tools such as easy-to-use I/O configuration for large-channel-count applications, automatic data logging, full alarm management and event logging, and real-time and historical trending.
- DIAdem is the interactive National Instruments software for finding and managing technical data, mathematically and graphically analyzing the data and presenting the data in reports.
- Biomedical Startup Kit is an add-on palette to LabVIEW designed for those interested in the fields of life science and biomedical engineering.
- LabVIEW Real-Time and RT Series hardware deliver deterministic, real-time control needed for machine monitoring and test systems.
- Typical systems in material handling include a vision system, belt drives as well as system specific sensors and actuators depending on system.
- A wide variety of tightly integrated DAQ, motion and real-time controllers are used to effectively solve the automation of the semiconductor lithography process.

---

## REVIEW QUESTIONS

---

1. List the various signal processing and analysis tools and their applications.
2. Explain how the professional development tools are used to optimize, test, and distribute VIs.
3. How are the third-party connectivity tools to Microsoft Office used for professional reporting, databases to access and store data, and embedded design tools, and control design and simulation tools.
4. Draw the block diagram and explain the three LabVIEW Control Design Tools.
5. What is the purpose of DIAdem software? List its applications.
6. Draw the block diagram and explain typical systems in material handling including a vision system, belt drives, sensors and actuators.
7. How does National Instruments offer a wide variety of tightly integrated DAQ, motion and real-time controllers to effectively solve the automation of the semiconductor lithography process.
8. Draw and explain the block diagram of plastic injection molding system.

# Index

---

- 3D curve graph, 141
- 3D graph, 131, 141
- 3D parametric surface graph, 141
- 3D surface graph, 141
  
- ADC, 265, 279
- Amplification, 258
- Analog input, 264, 265
- Analog output, 264, 266
- Area scan, 311
- Array functions, 98
- Array subset, 97
- Array to spreadsheet string, 197, 200
- Array, 91, 133
- ASCII, 204
- Auto indexing, 100, 101
- Autoscale, 142
  
- Backlash code, 202
- Binary, 204
- Block diagram, 24, 34
  
- Block diagram toolbar, 27
- Boolean controls and indicators, 33
- Build array, 106
- Build text, 197
- Build text Express VI, 198
- Build waveform, 135, 140
- Bundle by name, 120
- Bundle function, 115, 117, 118, 134
  
- Case selector, 162
- Case structure, 160
- Chart, 131, 133
- Cluster, 115
- Cluster constant, 117
- Cluster control, 116
- Cluster indicator, 116
- Cluster order, 117
- Code, 124
- Commercial platforms, 7
- Concatenate string, 196
- Condition terminal, 68
- Connector pane, 51

- Control design toolkit, 364, 375  
Controls, 33  
Controls palette, 29  
COTS, 5  
Count terminal, 66  
Counter, 264, 267  
Cursor legend, 143
- DAC, 264, 267  
DAQ, 21, 253  
DAQ assistant, 270  
DAQ card, 284  
DAQ device, 263  
DAQ hardware, 261  
DAQ signal accessory, 263  
DAQ software, 269  
Data flow program, 36  
Data types, 36  
Deployment, 4  
DIADem, 379  
Digital filter design toolkit, 362, 365  
Digital I/O, 264, 268  
Digital waveform graph, 131, 139  
Disable autoindexing, 102  
Driver, 269  
DSC module, 380
- .exe extension, 58  
Edge detection, 301, 302, 313  
Embedded module, 381  
Enable autoindexing, 102  
Encoder, 352, 358  
Error cluster, 115, 123  
Error handling, 122  
Event structure, 175  
Express VIs, 35
- Feedback device, 326, 351  
Feedback nodes, 75  
File I/O, 206  
Filter, 259  
Flat sequence, 163  
For loop, 65, 160  
Format into file, 202  
Format into string, 200, 201
- Format value, 201  
Formula node, 171  
Front panel, 23  
Front panel toolbar, 26  
Functions, 35  
Functions palette, 30
- G programming language, 1  
Gauging, 313  
Geometric matching, 301, 303  
Global variable, 81  
GPIB, 21, 223, 225  
Graph, 131  
Graph palette, 143  
Graphical system design (GSD), 1, 2, 16, 382
- Hex display, 202  
High level file I/O, 207
- Icon, 50  
Icon/connector pane, 25, 49  
IMAQ, 293  
Index array, 98, 99  
Indicators, 33  
Instrument driver, 232  
Instrument I/O assistant, 228  
Intensity chart, 136, 137  
Intensity graph, 136, 138  
Isolation, 258  
Iterative terminal, 66, 69
- Keyboard shortcuts, 38
- .lvm file, 204  
LabVIEW, 15, 20, 232  
Line scan, 311  
Linearization, 259  
Local variable, 79
- Machine vision, 301  
Match pattern, 196  
MathScript, 176

- 
- MATLAB script, 179  
 Matrix operation with array, 102, 112  
 MAX, 228, 260, 310, 327  
 Measurement system, 277, 282  
 Modular I/O, 7  
 Modular programming, 47  
 Modulation toolkit, 363  
 Motion assistant, 325  
 Motion control system, 325  
 Motion controller, 326, 330  
 Multiple plot, 132, 134, 145  
 Multiplexing, 258
- Nichols plane, 149  
 Nodes, 35  
 Normal display, 202  
 Numeric controls and indicators, 33  
 Nyquist plane, 149
- One-dimensional array, 96, 98
- Pallettes, 28  
 Particle analysis, 299, 313  
 Password display, 202  
 Pattern matching, 301, 303, 313  
 Plot legend, 143  
 Polymorphism, 104  
 Prototype, 3, 326, 328  
 PXI, 21, 223, 285
- Resolver, 355  
 RS 232, 236  
 RTSI, 265, 359
- S plane, 149  
 Scalar, 101, 104  
 Scale legend, 137, 143, 144, 146  
 Scan from file, 202  
 Scan from string, 197, 199, 201  
 Scan value, 201  
 Scope chart, 147  
 Sequence structure, 163  
 Serial port, 223, 235
- Servo amplifier, 335, 336  
 Servo motor, 337, 338  
 Shift register, 71  
 Signal conditioning, 256  
 Signal sources, 276  
 Signals, 254  
 Single plot, 132, 134, 145  
 Sound and vibration toolkit, 363, 367  
 Source, 124  
 Spectral measurement toolkit, 363  
 Spreadsheet string to array, 197, 200  
 Stack plot, 147, 148  
 Stacked shift register, 74  
 Status, 123
- Stepper motor, 335, 344  
 String controls and indicators, 33, 194  
 String function, 196  
 String length, 196  
 Strip chart, 147  
 Structure, 160, 166  
 SubVIs, 35, 49, 57  
 Sweep chart, 147  
 System identification toolkit, 365, 379
- Tables, 195  
 Tacked sequence, 163  
 TCP/IP, 21  
 Terminals, 34  
 Terminal block, 263  
 Textual programming, 17  
 Three-dimensional array, 97, 99, 101  
 Timed loop, 169  
 Timed structure, 168  
 Tools palette, 28  
 Traditional instrument, 7  
 Transducer, 254  
 Tunnel, 70, 74, 160, 162  
 Two-dimensional array, 93, 97, 99, 100, 101, 102
- Unbundle by name, 121  
 Unbundle cluster, 117, 118  
 USB, 223, 286
- Virtual instrumentation, 1, 6, 10, 12  
 VIs, 8, 54

- VISA, 223, 230  
Vision, 293, 313  
Vision assistant, 313
- Wait function, 55  
Wait until next ms multiple function, 77  
Waveform chart, 133  
Waveform graph, 131, 132  
While loop, 65, 67, 160  
Wires, 35
- Write LabVIEW measurement file, 208  
Write to spread sheet file, 207  
Write to/read from measurement file, 207
- Xmath script, 179  
XY graph, 131, 135
- Z plane, 149

# Virtual Instrumentation Using LabVIEW

**Jovitha Jerome**

This book provides a practical and accessible understanding of the fundamental principles of virtual instrumentation. It explains how to acquire, analyze and present data using LabVIEW (Laboratory Virtual Instrument Engineering Workbench) as the application development environment.

The book introduces the students to the graphical system design model and its different phases of functionality such as design, prototyping and deployment. It explains the basic concepts of graphical programming and highlights the features and techniques used in LabVIEW to create Virtual Instruments (VIs). Using the technique of modular programming, the book teaches how to make a VI as a subVI. Arrays, clusters, structures and strings in LabVIEW are covered in detail. The book also includes coverage of emerging graphical system design technologies for real-world applications. In addition, extensive discussions on data acquisition, image acquisition, motion control and LabVIEW tools are presented.

This book is designed for undergraduate and postgraduate students of instrumentation and control engineering, electronics and instrumentation engineering, electrical and electronics engineering, electronics and communication engineering, and computer science and engineering. It will be also useful to engineering students of other disciplines where courses in virtual instrumentation are offered.

## KEY FEATURES

- ◆ Builds the concept of virtual instrumentation by using clear-cut programming elements.
- ◆ Includes a summary that outlines important learning points and skills taught in the chapter.
- ◆ Offers a number of solved problems to help students gain hands-on experience of problem solving.
- ◆ Provides several chapter-end questions and problems to assist students in reinforcing their knowledge.

## ABOUT THE AUTHOR

**JOVITHA JEROME** (D. Eng. from Asian Institute of Technology, Bangkok) is Professor and Head, Department of Instrumentation and Control Systems Engineering, PSG College of Technology, Coimbatore. She is also Professor In-charge of PSG-NI Virtual Instrumentation Centre of the Institute. In this capacity, she received the Best Graphical System Design Lab Award (2008) from National Instruments (NI), Bangalore. She has about three decades of teaching experience. Earlier, she taught at Government College of Technology, Coimbatore; Coimbatore Institute of Technology, Coimbatore; and Sirindhorn International Institute of Technology (SIIT), Bangkok. Dr. Jovitha Jerome has published several papers in international and national journals. She received the Women Engineer Award (2008) from the Tamil Nadu State Centre of the Institution of Engineers (India) and the Outstanding Engineer Award (2009) from the Coimbatore Local Centre of the Institution of Engineers (India).

**Rs. 375.00**

[www.phindia.com](http://www.phindia.com)

