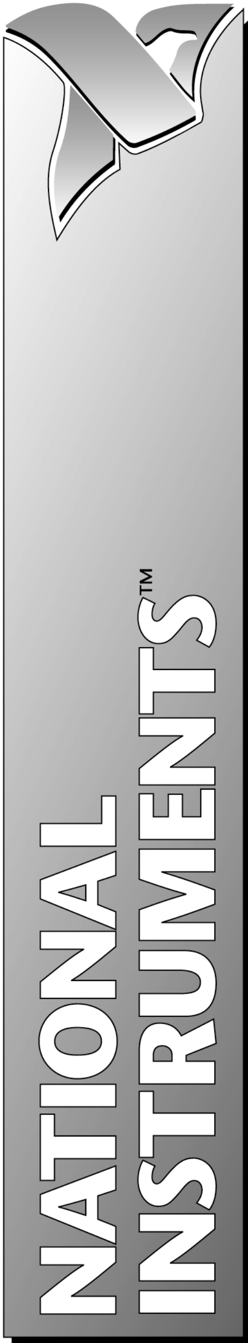


Professional G Developers Toolkit Reference Manual

January 1997 Edition
Part Number 321393A-01

© Copyright 1997 National Instruments Corporation. All rights reserved.

Internet Support

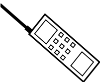
support@natinst.com
E-mail: info@natinst.com
FTP Site: ftp.natinst.com
Web Address: http://www.natinst.com

Bulletin Board Support

BBS United States: (512) 794-5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

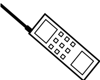
Fax-on-Demand Support

(512) 418-1111



Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678



International Offices

Australia 02 9874 4100, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,
Finland 09 527 2321, France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186,
Israel 03 5734815, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456,
Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886,
Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200,
U.K. 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway   Austin, TX 78730-5039   Tel: (512) 794-0100

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

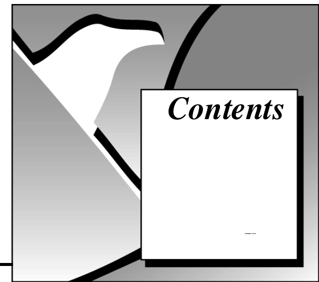
Trademarks

LabVIEW® and BridgeVIEW™ are trademarks of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.



About This Manual

Organization of This Manual xiii
Software Engineering Concepts xiii
Professional Development Tools xiv
Appendices, Glossary, and Index xiv
Conventions Used in This Manual xv
Related Documentation xvi
Customer Communication xvi

Chapter 1
Introduction

Required System Configuration 1-1
Installation and Configuration 1-2
Installation 1-2
Configuration 1-3
Overview 1-3
Features of the Toolkit 1-4

Chapter 2
Development Models

Common Development Pitfalls 2-1
Lifecycle Models 2-4
Code and Fix Model 2-5
Waterfall Model 2-5
Modified Waterfall Model 2-8
Prototyping 2-8
G Prototyping Methods 2-9
Spiral Model 2-10
Summary 2-12

Chapter 3
Incorporating Quality into the Development Process

Quality Requirements 3-1

Configuration Management 3-2
Source Code Control 3-2
Managing All Project-Related Files 3-3
Retrieving Old Versions of Files 3-3
Tracking Changes 3-4
Change Control 3-4
Testing Guidelines 3-5
Black Box and White Box Testing 3-6
Unit, Integration, and System Testing 3-7
Unit Testing 3-7
Integration Testing 3-8
System Testing 3-9
Formal Methods of Verification 3-10
Style Guidelines 3-10
Design Reviews 3-11
Code Walkthroughs 3-12
Postmortem Evaluation 3-13
Software Quality Standards 3-13
International Organization for Standards (ISO) 9000 3-13
U.S. Food & Drug Administration (FDA) Standards 3-14
Capability Maturity Model (CMM) 3-15
Institute of Electrical and Electronic Engineers (IEEE) Standards 3-16

Chapter 4
Prototyping and Design Techniques

Clearly Define the Requirements of Your Application 4-1
Top-Down Design 4-2
Example—A Data Acquisition System 4-4
Bottom-Up Design 4-6
Example—An Instrument Driver 4-7
Designing for Multiple Developers 4-9
Front Panel Prototyping 4-10
Performance Benchmarking 4-11
Identify Common Operations 4-11

Chapter 5
Scheduling and Project Tracking

Estimation 5-1
Lines of Code/Number of Nodes Estimation 5-2
Problems with Lines of Code and Numbers of Nodes 5-3
Effort Estimation 5-4
Wideband Delphi Estimation 5-4

Other Estimation Techniques 5-5
Mapping Estimates to Schedules 5-6
Tracking Schedules Using Milestones 5-7
Responding to Missed Milestones 5-8

Chapter 6
Creating Documentation

Developing Design-Related Documentation 6-1
Developing User Documentation 6-2
Documentation for a Library of SubVIs 6-2
Documentation for an Application 6-3
Creating Help Files 6-4
VI and Control Descriptions 6-5
VI Description 6-5
Self-Documenting Front Panels 6-5
Control and Indicator Descriptions 6-6

Chapter 7
Using Consistent Style—The G Style Guide

Hierarchy on Disk 7-1
Hierarchy with VI Libraries 7-3
Front Panels with Style 7-4
Consistency 7-4
Text 7-5
Color 7-6
Graphics and Custom Controls 7-6
Front Panel Layout 7-7
Sizing and Positioning Front Panels 7-8
Controls and Indicators 7-8
Descriptions 7-8
Labels 7-8
Enumerations vs. Rings 7-9
Default Values, Ranges, and Coercion 7-10
Attribute Nodes 7-11
Key Navigation 7-11
Local Variables 7-12
VI Setup 7-13
Connector Panes 7-14
Icons 7-15
The Block Diagram 7-17
Wiring Etiquette 7-17
Labeling 7-18

Execution Sequence 7-19
Left-to-Right Layouts 7-19
Data Dependency 7-19
Adding Common Threads 7-19
Sequence Structures 7-20
Watch Out for Missing Dependencies 7-21
Check for Errors 7-22
Sizing and Positioning of Block Diagrams 7-23
Optimization 7-24
Code Interface Nodes 7-25
CIN Description Contents 7-25
CIN Source Code 7-25
Style Checklist 7-26
VI Checklist 7-26
Front Panel Checklist 7-27
Block Diagram Checklist 7-29

Chapter 8
VI Metrics Tool

Additional Statistics 8-3
Block Diagram Statistics 8-3
User Interface Statistics 8-4
Global/Local Statistics 8-4
CIN/Shared Library Statistics 8-4
SubVI Interface Statistics 8-5
Files in vi.lib 8-5
Saving Metric Information 8-5

Chapter 9
Print Hierarchy Tool

Chapter 10
File Manager Tool

Chapter 11
Source Code Control Tools

General Source Code Control Concepts 11-1
Using Individual Files Instead of VI Libraries (LLBs) 11-2
Source Code Control Configuration 11-2
Selecting the Source Code Control System 11-3

Features of the Built-In Source Code Control System........................ 11-3
Features of Third-Party Source Code Control Systems...................... 11-4
Microsoft Visual SourceSafe for Windows 95/NT 11-5
Administration (Administrator Only).. 11-5
Administration of Visual SourceSafe 11-6
Administration of the Built-In System... 11-7
Edit Platform List.. 11-8
Local Configuration (All Users)... 11-9
User Configuration of Visual SourceSafe.. 11-9
User Configuration of the Built-In System 11-9
Work Directory and Platform Configuration...................................... 11-10
Managing Source Code Control Projects.. 11-10
Source Code Control Projects Overview... 11-10
Creating a Project 11-11
Updating a Project 11-13
Removing Files from a Project... 11-13
Adding Extra Files to a Project 11-14
Project Groups 11-15
Accessing Files 11-16
Retrieving Files 11-16
File Status.. 11-17
File Properties 11-18
Checking Out Files 11-18
Use the History Window to Document Changes 11-20
Checking In Files.. 11-20
SCC User Name 11-22
Advanced Features.. 11-22
Deleting Files from SCC 11-23
SCC File History 11-24
System History 11-25
Master File List (sccfiles.lst).. 11-25
Accessing Previous Versions of Files 11-26
Built-In System 11-26
Third-Party Systems.. 11-27
Labeling Versions of Files for Easy Retrieval 11-27
Creating Reports 11-27
Built-In System 11-28
Microsoft Visual SourceSafe 11-29
Multiplatform Issues 11-29
Cross Platform Source Code Control 11-29
Filename Limitations... 11-29
Platform-Dependent SCC Files.. 11-30
Platform-Specific Files 11-31

Variants of a File for Different Platforms.. 11-31
Retrieving Files for a Different Platform 11-32

Appendix A
References

Appendix B
Customer Communication

Glossary

Index

Figures

Figure 2-1. The Waterfall Model 2-6
Figure 2-2. The Spiral Model 2-10

Figure 3-1. Capability Maturity Model 3-16

Figure 4-1. Mapping pseudo-code into a G data structure. 4-5
Figure 4-2. Mapping pseudo-code into actual G code. 4-5
Figure 4-3. Data Flow for a Generic Data Acquisition Program 4-6
Figure 4-4. VI hierarchy for the Tektronix 370A 4-8

Figure 7-1. A directory hierarchy. 7-2
Figure 7-2. Top-Level VIs Listed at the Top of a VI Library 7-3
Figure 7-3. A mixture of directories and VI libraries. 7-4
Figure 7-4. Good Wiring in a Simple Block Diagram 7-18
Figure 7-5. A Well-Placed Front Panel and Block Diagram 7-24

Figure 8-1. VI Metrics Tool Dialog Box 8-1

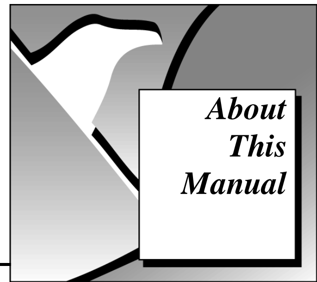
Figure 9-1. Print Hierarchy Tool Dialog Box 9-1

Figure 10-1. File Manager Tool Dialog Box 10-1

Figure 11-1. G Source Code Control Tools Work with Built-In and
Third-Party Systems 11-3

Tables

Table 2-1. Risk Exposure Analysis Example 2-11



The *Professional G Developers Toolkit Reference Manual* describes the features, functions, and operation of the Professional G Developers Toolkit. With this toolkit, you can apply software engineering techniques to G code development. This toolkit includes software extensions to BridgeVIEW and LabVIEW that provide important software engineering tools.

In addition, this manual describes many of the issues that arise when developing large applications and provides a basic survey of software engineering techniques you might find useful when developing your own projects.

Organization of This Manual

The *Professional G Developers Toolkit Reference Manual* is divided into two sections. Chapters 1 through 7 describe software engineering concepts. Chapters 8 through 11 describe the tools included with this toolkit.

Software Engineering Concepts

- Chapter 1, *Introduction*, describes the installation procedure and introduces you to the features of the Professional G Developers Toolkit.
- Chapter 2, *Development Models*, provides examples of some common development pitfalls and describes a number of software engineering lifecycle models.
- Chapter 3, *Incorporating Quality into the Development Process*, describes strategies for producing quality software.
- Chapter 4, *Prototyping and Design Techniques*, gives you pointers for project design, including programming approaches, prototyping, and benchmarking.
- Chapter 5, *Scheduling and Project Tracking*, describes techniques for developing estimates of development time and using those estimates to create schedules.

- Chapter 6, *Creating Documentation*, describes techniques for documenting your software.
- Chapter 7, *Using Consistent Style—The G Style Guide*, describes some recommended practices for good programming technique and style.

Professional Development Tools

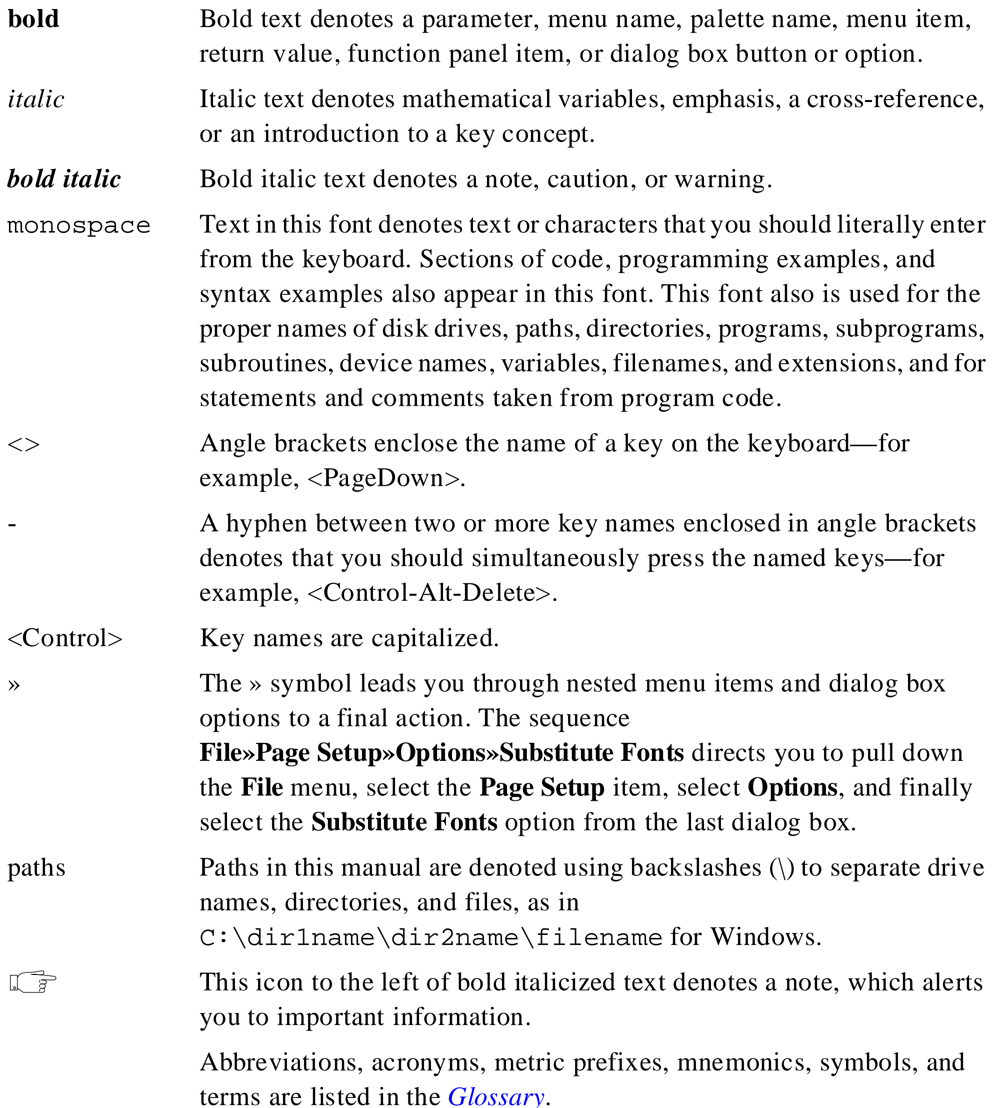
- Chapter 8, *VI Metrics Tool*, describes how to use the VI Metrics tool to measure the complexity of your application.
- Chapter 9, *Print Hierarchy Tool*, describes how the Print Hierarchy tool makes it easy to print out VI documentation for VIs in your hierarchy.
- Chapter 10, *File Manager Tool*, describes how the File Manager tool makes it possible to easily copy, rename, or delete files within VI Libraries (LLBs).
- Chapter 11, *Source Code Control Tools*, describes the G Source Code Control tools.

Appendices, Glossary, and Index

- Appendix A, *References*, provides a list of references for further information about software engineering concepts.
- Appendix B, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:



bold	Bold text denotes a parameter, menu name, palette name, menu item, return value, function panel item, or dialog box button or option.
italic	Italic text denotes mathematical variables, emphasis, a cross-reference, or an introduction to a key concept.
bold italic	Bold italic text denotes a note, caution, or warning.
`monospace`	Text in this font denotes text or characters that you should literally enter from the keyboard. Sections of code, programming examples, and syntax examples also appear in this font. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, variables, filenames, and extensions, and for statements and comments taken from program code.
<>	Angle brackets enclose the name of a key on the keyboard—for example, <PageDown>.
-	A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Control-Alt-Delete>.
<Control>	Key names are capitalized.
»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options»Substitute Fonts** directs you to pull down the **File** menu, select the **Page Setup** item, select **Options**, and finally select the **Substitute Fonts** option from the last dialog box.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, and files, as in `C:\dir1name\dir2name\filename` for Windows.
☞	This icon to the left of bold italicized text denotes a note, which alerts you to important information.

Abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms are listed in the *Glossary*.

Related Documentation

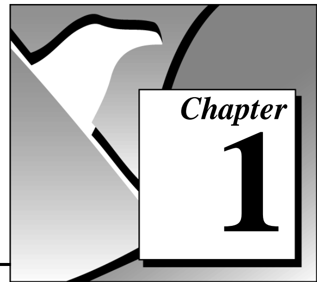
The following documentation contains information that you might find helpful as you read this manual.

- *BridgeVIEW User Manual*
- *LabVIEW User Manual*
- *G Programming Reference Manual*
- BridgeVIEW *Online Reference*, available by selecting **Help»Online Reference**
- LabVIEW *Online Reference*, available by selecting **Help»Online Reference**

See Appendix A, *References*, for a list of additional documents you might find helpful as you read this manual and work on your development projects.

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix B, *Customer Communication*, at the end of this manual.



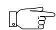
Introduction

This chapter describes the installation procedure and introduces you to the features of the Professional G Developers Toolkit.

Required System Configuration

The Professional G Developers Toolkit works with LabVIEW 4.0.1 or BridgeVIEW 1.0. If you have an older version of LabVIEW, you need to upgrade. If you have LabVIEW 4.0, the upgrade to 4.0.1 is free. Contact National Instruments for information on upgrading to the latest version.

This toolkit is available for all platforms except Windows 3.1. Windows 3.1 is not supported because the Source Code Control (SCC) tools require that virtual instruments (VIs) be stored in individual files rather than in libraries (LLBs). You can use the File Manager tool to convert LLBs to directories.

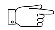
 **Note:** ***While you cannot use this toolkit under Windows 3.1, you still can develop for customers who need Windows 3.1 support. You can use Windows 95 or NT as your development platform and save your VIs in individual files. When you need to send software to a Windows 3.1 user, save your VIs in LLBs.***

The SCC tools cannot manage multiple files with the same name. In fact, the tools help prevent you from using files with the same name. This is to help you avoid accidentally linking a VI to a subVI with the same name.

Installation and Configuration

You can install the Professional G Developers Toolkit on the following platforms:

- Windows 95 and NT
- Macintosh and Power Macintosh
- Solaris for Sun SPARCstation
- HP-UX

 **Note:** ***Some virus detection programs might interfere with the installer program. Turn off the automatic virus checker before running the installer. After installation, check your hard disk for viruses and turn on the automatic virus checker.***

Installation

The Professional G Developers Toolkit CD contains the installer programs for all of the platforms listed above. Run the installer program for your platform. The following list includes the names of the setup files and the directories in which they are located.

- **(Windows 95 and NT**) `setup.exe` in the `win95_nt` directory
- **(68K Macintosh)** `ProDev 68K Installer` in the `68K Macintosh` folder
- **(Power Macintosh)** `ProDev PM Installer` in the `Power Macintosh` folder
- **(Solaris)** `install` in the `solaris` directory
- **(HP-UX)** `INSTALL` in the `HPUX` directory

When prompted for the destination directory, select the directory in which you have BridgeVIEW 1.0 or LabVIEW 4.01 or later installed.

Also, to use the Source Code Control tools, you must decide if you want to install as a user or as an administrator. Usually only one person from a development team should be designated as the administrator. The administrator installation includes everything the user installation does, with the addition of the **Source Code Control»Administration** command that allows the administrator to set up the SCC system for use by the development team.

Configuration

After installation is complete, the administrator must set up the Source Code Control system for the other users. All users must perform local configuration. See the *Administration (Administrator Only)* and *Local Configuration (All Users)* sections in Chapter 11, *Source Code Control Tools* for more details.

Overview

LabVIEW and BridgeVIEW are flexible tools for designing test, measurement, and process monitoring and control applications with a graphical programming language called G.

National Instruments designed G as an easy-to-use and general purpose language. Because G is a fully functional programming language, LabVIEW and BridgeVIEW handle complex applications that cannot be developed easily using more restrictive data acquisition and control applications. G emphasizes hierarchical design and reuse with its concept of a VI. Each VI is a complete program consisting of a front panel, which provides a user interface, and a block diagram, which represents the source code. The block diagram describes the relationship and interactions between inputs and outputs in the user interface. Every VI can be a reusable component because you can define a calling interface and a representative icon so the VI can be called as a subroutine, or subVI, from other VIs.

Because VIs can be stored in separate files, it is possible to have multiple developers work on different parts of a project simultaneously. As with other programming languages, developing and managing large applications with multiple developers requires more rigorous methodologies than are required with simple applications. Poor design and development techniques can lead to applications that are not developed on time, are not easy to maintain, and contain programming errors that prevent the software from working reliably.

Software engineering is the field of study related to defining the best processes for developing software, and the main goal of this toolkit is to help users apply these techniques to G code development. Most of the techniques developed in software engineering apply to graphical programming languages just as well as they apply to textual programming languages.

Features of the Toolkit

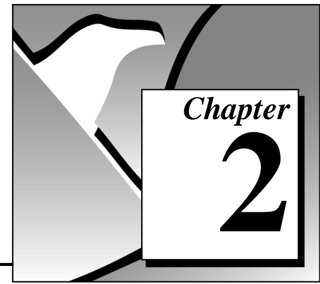
The Professional G Developers Toolkit is designed to simplify development of high-end, large-scale applications. It includes tools for managing and tracking code in large development projects. These tools are ideal for large teams of developers, individual users developing large suites of VIs, and G programmers who must adhere to stringent quality standards such as those required by ISO 9000 or FDA.

The toolkit includes features that help you do the following:

- Control source code—Integrated Source Code Control (SCC) tools are accessible from the menus of LabVIEW or BridgeVIEW. With these tools, you can share VIs with multiple developers.

  You can check out files to begin development and check in files when you are ready to share your changes with others. This check out/check in system ensures that only one developer modifies a specific VI at a time. The G SCC tools are built on an open Application Programming Interface (API) so that they can communicate with either a built-in SCC system available to all platforms or other third-party SCC systems. This toolkit supports the built-in system and Microsoft Visual SourceSafe for Windows 95/NT.
- Measure complexity—The VI Metrics tool provides a simple way to measure the complexity of an application similar to the widely-used Source Lines of Code (SLOCs) metrics for textual languages. With this tool, you also can view many other statistics about VIs, all of which are useful in either examining your VIs to find overly complex areas or in establishing baselines for estimating future projects.
- Print documentation—The Print Hierarchy tool makes it easy to print out documentation for VIs in your hierarchy. This tool provides a lot of flexibility in printing out VI documentation in a variety of formats.
- Edit VI libraries—The File Manager tool automates the process of copying, renaming, or deleting files within VI Libraries (LLBs). You can use this tool to convert LLBs to directories, an important step toward managing your VIs with the Source Code Control tools.

Development Models



This chapter provides examples of some common development pitfalls and describes a number of software engineering lifecycle models.

G, the graphical programming language of LabVIEW and BridgeVIEW, makes it easy to assemble components of data acquisition, test, and control systems. Because it is so easy to program in G, you might be tempted to begin developing VIs immediately with relatively little planning. For very simple applications, such as quick lab tests or monitoring applications, this approach might be appropriate. However, for larger development projects, good planning becomes vital.

Common Development Pitfalls

If you have developed large applications before, you probably have heard some of the following quotes. Most of these approaches start out with good intentions and seem quite reasonable. However, these approaches are often unrealistic and can lead to delays, quality problems, and poor morale among team members.

- "I haven't really thought it through, but I'd guess that the project you are requesting can be completed in…"

  Off-the-cuff estimates rarely are correct because they usually are based on an incomplete understanding of the problem. When developing for someone else, you might each have different ideas about requirements. To estimate accurately, you both must clearly understand the requirements and work through at least a preliminary high-level design so you understand the components you need to develop. Techniques for estimation are described in more detail in Chapter 5, *Scheduling and Project Tracking*.

- "I think I understand the problem the customer wants to solve, so I'm ready to dive into development."

  There are two problems with a statement like this. First, lack of consensus on project goals results in schedule delays. Your idea of what a customer wants might be based on inadequate communication. Developing a requirements document and prototyping a system, both described later, can be useful tools to clarify goals. A second problem with this statement is that diving into development might mean writing code without a detailed design. Just as builders do not construct a building without architectural plans, developers should not begin building an application without a detailed design. See the *Code and Fix Model* section later in this chapter for more information.

- "We don't have time to write detailed plans—we're under a tight schedule, so we need to start developing right away."

  This situation is similar to the previous example, but is such a common mistake that it is worth emphasizing. Software developers frequently skip important planning because it does not seem as productive as developing code. As a result, you develop VIs without a clear idea of how they all fit together, and you might have to rework sections as you discover mistakes. Taking the time to develop a plan can prevent costly rework at the development stage. See the *Lifecycle Models* section later in this chapter and Chapter 4, *Prototyping and Design Techniques*, for better approaches to developing software.

- "Let's try for the whole ball of wax on the first release—if it doesn't do everything, it won't be useful."

  In some cases, this might be correct. However, in most applications, developing in stages is a better approach. When analyzing the requirements for a project, you should prioritize features. You might be able to develop an initial system that provides useful functionality in a shorter time at a lower cost. Then, you can add features incrementally. The more you try to accomplish in a single stage, the greater the risk of falling behind schedule. Releasing software incrementally reduces schedule pressures and ensures timely software release. See the *Lifecycle Models* section later in this chapter for more information.

- "If I can just get all of the features in within the next month, I should be able to fix any problems before the software is released."

  To release high-quality products on time, maintain quality standards throughout development. Do not build new features on an

unstable foundation and rely on correcting problems later. This exacerbates problems and increases cost. While you might complete all of the features on time, the time required to correct the problems in both the existing and new code can delay the release of the product. You should prioritize features and implement the most important ones first. Once the most important features are tested thoroughly, you can choose to work on lower-priority features or defer them to a future release. See Chapter 3, *Incorporating Quality into the Development Process*, for more details on techniques for producing high-quality software.

- "We're behind in our project—let's throw more developers onto the problem."

  In many cases, doing this actually can delay your project. Adding developers to a project requires time for training, which can take away time originally scheduled for development. Add resources earlier in the project rather than later. Also, there is a limit to the number of people who can work on a project effectively. With a few people, there is less overlap—you partition the project so each person works on a particular section. The more people you add, the more difficult it becomes to avoid overlap. Chapter 4, *Prototyping and Design Techniques*, describes methods for partitioning software for multiple developers. Chapter 3, *Incorporating Quality into the Development Process*, describes configuration management techniques that can help minimize overlap.

- "We're behind in our project, but we still think we can get all the features in by the specified date."

  When you are behind in a project, it is important to recognize that fact and deal with it. Assumptions that you can make up lost time can defer choices until it becomes costly to deal with them. For example, if you realize in the first month of a six-month project that you are behind, you could sacrifice planned features or add time to the overall schedule. If, in the fifth month, you find the schedule slipping, other groups might have made decisions that are costly to change.

  When you realize you are behind, consider features that can be dropped or postponed to subsequent releases, or adjust the schedule. Do not ignore the delay or sacrifice testing scheduled for later in the process. Estimating project schedules is described in more detail in Chapter 5, *Scheduling and Project Tracking*.

Numerous other problems can arise when developing software. The following list includes some of the fundamental elements of developing quality software on time:

- Spend sufficient time planning.
- Make sure the whole team thoroughly understands the problems that must be solved.
- Have a flexible development strategy that minimizes risk and accommodates changes.

Lifecycle Models

Software development projects are complex. To deal with these complexities, developers have collected a core set of development principles. These principles define the field of software engineering. A major component of this field is the *lifecycle model*. The lifecycle model describes the steps you follow to develop software—from the initial concept stage to the release, maintenance, and subsequent upgrading of the software.

Currently, there are many different lifecycle models. Each has its own advantages and disadvantages in terms of time-to-release, quality, and risk management. This section describes some of the most common models used in software engineering. Many hybrids of these models exist, so use the parts you believe will work for your project.

While this section is theoretical in its discussion, in practice you should consider all of the steps that these models encompass. You should consider when and how you decide that the requirements and specifications are complete and how you handle changes to them. The lifecycle model serves as a foundation for the entire development process. Good choices in this area can improve the quality of the software you develop and decrease the time it takes to develop it.

Code and Fix Model

The *code and fix model* probably is the most frequently used development methodology in software engineering. It starts with little or no initial planning. You immediately start developing, fixing problems as you find them, until the project is complete.

Code and fix is a tempting choice when you are faced with a tight development schedule because you begin developing code right away and see immediate results.

Unfortunately, if you find major architectural problems late in the process, you might have to rewrite large parts of your application. Alternative development models can help you catch these problems in the early concept stages when it is much less expensive to make changes. Accommodating changes in the requirements and specifications stage of a project is easier than after you have written a lot of code.

The code and fix model is only appropriate for small projects that are not intended to serve as the basis for future development.

Waterfall Model

The *waterfall model* is the classic model of software engineering. It has deficiencies, but it serves as a baseline for many other lifecycle models.

The pure waterfall lifecycle consists of several non-overlapping stages, which are listed below. It begins with the software concept and continues through requirements analysis, architectural design, detailed design, coding, testing, and maintenance. Figure 2-1 illustrates the stages of the waterfall lifecycle model.

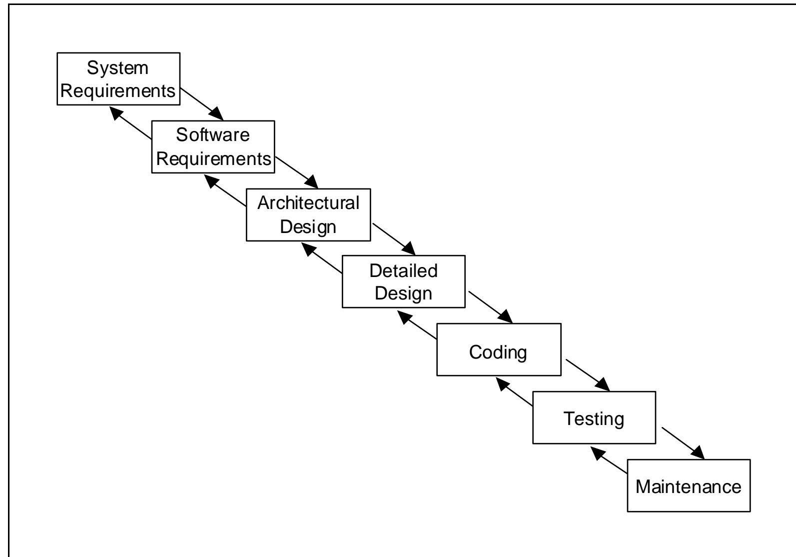


Figure 2-1. The Waterfall Model

- System requirements—Establishes the components for building the system. This includes the hardware requirements (number of channels, acquisition speed, and so on), software tools, and other necessary components.
- Software requirements—Concentrates on the expectations for software functionality. You identify which of the system requirements are affected by the software. Requirements analysis might include determining interaction needed with other applications and databases, performance requirements, user interface requirements, and so on.
- Architectural design—Determines the software framework of a system to meet the specified requirements. The design defines the major components and their interaction, but it does not define the structure of each component. In addition to the software framework, in the architectural design phase, you determine the external interfaces and tools that will be used in the project. Examples include decisions on hardware, such as plug-in boards, and external pieces of software, such as databases or other libraries.
- Detailed design—Examines the software components defined in the architectural design stage and produces a specification for how each component is implemented.

- Coding—Implements the detailed design specification.
- Testing—Determines whether the software meets the specified requirements and finds any errors present in the code.
- Maintenance—Performed as needed to address problems and enhancement requests after the software is released. In some organizations, each change is reviewed by a change control board to ensure that quality is maintained. You also can apply the full waterfall development cycle model when you implement these change requests.

In each stage, you create documents that explain your objectives and describe the requirements for that phase. At the end of each stage, you hold a review to determine whether the project can proceed to the next stage. Also, you can incorporate prototyping into any stage from the architectural design and after. See the *Prototyping* section later in this chapter for more information.

The waterfall lifecycle model is one of the oldest models and is widely used in government projects and in many major companies. Because it emphasizes planning in the early stages, it helps catch design flaws before they are developed. Also, because it is very document- and planning-intensive, it works well for projects in which quality control is a major concern.

Many people believe you should not apply this model to all situations. For example, with the pure waterfall model you must state the requirements before beginning the design, and you must state the complete design before you begin coding. That is, there is no overlap between stages. In real-world development, however, you might discover issues during the design or coding stages that point out errors or gaps in the requirements.

The waterfall method does not prohibit returning to an earlier phase (for example, from the design phase to the requirements phase). However, this involves costly rework—each completed phase requires formal review and extensive documentation development. Thus, oversights made in the requirements phase are expensive to correct later.

Because the actual development comes late in the process, you do not see results for a long time. This can be disconcerting to management and customers. Many people also find the amount of documentation is excessive and inflexible.

While the waterfall model has its weaknesses, it is instructive because it emphasizes important stages of project development. Even if you do not apply this model, you at least should consider each of these stages and its relationship to your own project.

Modified Waterfall Model

Many engineers recommend modified versions of the waterfall lifecycle. These modifications tend to focus on allowing some of the stages to overlap, reducing the documentation requirements, and reducing the cost of returning to earlier stages to revise them. Another common modification is to incorporate prototyping into the requirements phases, as described in the next section.

Overlapping stages such as requirements and design make it possible to feed information from the design phase back into the requirements. However, this can make it more difficult to know when you are finished with a given stage, and, consequently, it is more difficult to track progress. Without distinct stages, problems might cause you to defer important decisions until late in the process when they are more expensive to correct.

Prototyping

One of the main problems with the waterfall model is that the requirements are often not completely understood in the early development stages. When you reach the design or coding stages, you begin to see how everything works together, and you might discover that you need to adjust requirements.

Prototyping can be an effective tool for demonstrating how a design might address a set of requirements. You can build a prototype, then adjust the requirements and revise the prototype several times until you have a clear picture of your overall objectives. In addition to clarifying the requirements, the prototype also defines many areas of the design simultaneously.

The pure waterfall model does allow for prototyping in the later architectural design stage and subsequent stages, but not in the early requirements stages.

There are drawbacks with prototyping. First, because it appears that you have a working system very quickly, customers might expect a complete system sooner than is possible. In most cases, the prototype is built on compromises that allow it to come together quickly, but could

prevent the prototype from being an effective basis for future development. You need to decide early whether you will use the prototype as a basis for future development. All parties should agree with this decision before development begins.

You should be careful that prototyping does not become a disguise for a code and fix development cycle. Before you begin prototyping, you should gather clear requirements and create a design plan. Limit the amount of time you will spend prototyping before you begin. This helps to avoid overdoing the prototyping phase. As you incorporate changes, you should update the requirements and the current design. After you finish prototyping, you might consider falling back to one of the other development models. For example, you might consider prototyping as part of the requirements or design phases of the waterfall model.

G Prototyping Methods

There are a number of ways to prototype a system.

In systems with I/O requirements that might be difficult to satisfy, you can develop a prototype to test the control and acquisition loops and rates.

Systems with many user interface requirements are perfect for prototyping. Determining the method you will use to display data or prompt the user for settings can be difficult on paper. Instead, consider designing VI front panels with the controls and indicators you need. You might leave the block diagram empty and just talk through the way the controls would work and how various actions would lead to other front panels. For more extensive prototypes, you could even tie the front panels together; however be careful not to get too carried away with this process. In I/O prototypes, random data can simulate data acquired in the real system.

If you are bidding on a project for a client, this can be an extremely effective way to discuss with the client how you might be able to satisfy his or her requirements. Because you can add and remove controls quickly (especially if you avoid developing block diagrams), you can help customers clarify their requirements.

Spiral Model

The *spiral model* is a popular alternative to the waterfall model. It emphasizes risk management so you find major problems earlier in the development cycle. In the waterfall model, you have to complete the design before you begin coding. With the spiral model, you break up the project into a set of risks that need to be handled. You then begin a series of iterations in which you analyze the most important risk, evaluate options for resolving the risk, address the risk, assess the results, and plan for the next iteration. Figure 2-2 illustrates the spiral lifecycle model.

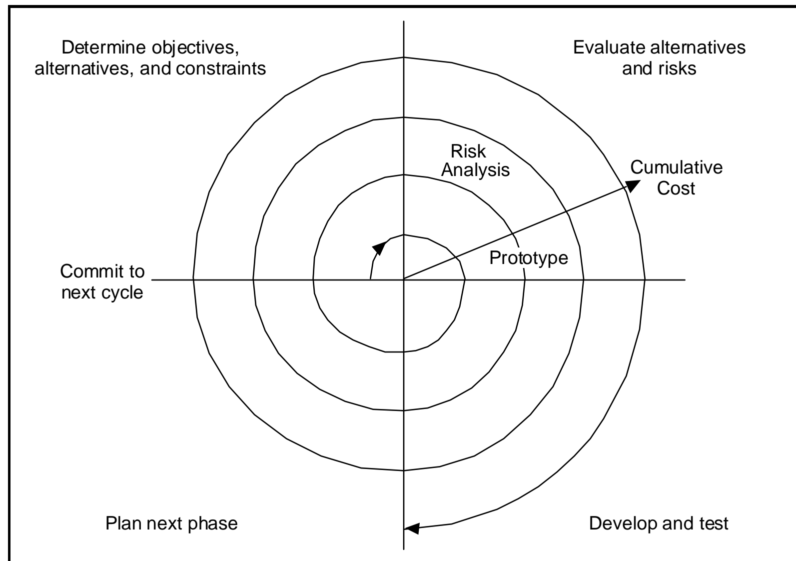


Figure 2-2. The Spiral Model

Risks are any issues that are not clearly defined or have the potential to affect the project adversely. For each risk, you need to consider two things: How likely it is, and how bad it is for it to occur. You might use a scale of 1 to 10 for each of these, with 1 being unlikely to occur and not bad if it occurs, and 10 being extremely likely to occur and catastrophic to the project if it occurs. Your risk exposure is the product of these. You can use a table to keep track of the top risk items of your project. See Table 2-1 for an example of how to do this.

Table 2-1. Risk Exposure Analysis Example

ID	Risk	Probability	Loss	Risk Exposure	Risk Management Approach
1	Acquisition rates too high	5	9	45	Develop prototype to demonstrate feasibility
2	File format may not be efficient	5	3	15	Develop benchmarks to show speed of data manipulation
3	Uncertain user interface	2	5	10	Involve customer, develop prototype

In general, you should address the risks with the highest risk exposure first. In this example, the first spiral should address the potential for the data acquisition rates being too high to handle. After the first spiral, you may have demonstrated that the rates are not too high, or you might have to change to a different configuration of hardware to meet the acquisition requirements. Each iteration might identify new risks. In this example, using more powerful hardware might make high cost a new, more likely, risk.

For example, assume that you are designing a data acquisition system with a plug-in data acquisition card. In this case, the risk is whether the system can acquire, analyze, and display data quickly enough. Some of the constraints in this case are requirements for a specific sampling rate and precision, and system cost.

After determining the options and constraints, you evaluate the risks. In this example, you could create a prototype or benchmark to test acquisition rates. After you see the results, you can evaluate whether to continue with the approach or choose a different option. You do this by reassessing the risks based on the new knowledge you have gained from building the prototype.

In the final phase, you evaluate the results with the customer. Based on customer input, you can reassess the situation, decide on the next highest risk, and start the cycle over. This process continues until the software is finished or you decide the risks are too great and terminate development. You might find that none of the options is viable because each is too expensive, time-consuming, or does not meet the requirements.

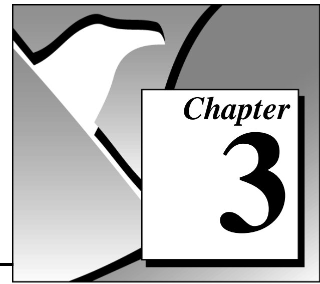
The advantage of the spiral model over the waterfall model is that you can evaluate which risks to address with each cycle. Because you can evaluate risks with prototypes much earlier than in the waterfall process, you can address major obstacles and select alternatives in the earlier stages, which is less expensive. With a standard waterfall model, you might have allowed assumptions about the risky components to spread throughout your design, requiring much more expensive rework when the problems are later discovered.

Summary

Lifecycle models are described as distinct choices from which you must select. In practice, however, you can apply more than one model to a single project. You might start a project with a spiral model to help refine the requirements and specification over several iterations using prototyping. Once you have reduced the risk of a poorly stated set of requirements, you might apply a waterfall lifecycle model to the design, coding, testing, and maintenance stages.

There are other lifecycle models not presented here. If you are interested in exploring other development methodologies, refer to Appendix A, *References*.

Incorporating Quality into the Development Process



This chapter describes strategies for producing quality software.

Many developers who follow the code and fix style of programming described in Chapter 2, *Development Models*, mistakenly believe that the issue of quality does not need to be addressed until the testing phase. This is simply not true. Quality must be designed into a product from the start. Developing quality software begins by selecting a development model that helps you avoid problems in the first place. Quality should be considered during all stages of development—requirements and specification, design, coding, testing, release, and maintenance.

Quality controls should not be regarded as tedious requirements that impede development. Most of them help streamline development so that problems are found when it is inexpensive to fix them—before they are in the software.

Quality Requirements

During the requirements stage, first set the quality standards for your product. The desired quality level should be treated as a requirement, just like other requirements. Weigh the merits and costs of various options you have for applying quality measures to your project. Some of the trade-offs you should consider are speed versus robustness, and ease-of-use versus power and complexity.

For short projects that only will be used in-house as tools or quick prototypes, you do not need to emphasize robustness. For example, if you decide to develop a VI to benchmark I/O and graphing speeds, little error checking is necessary.

For more complicated projects that must be reliable, such as applications for monitoring and controlling a factory process, the software should handle invalid input gracefully. For example, if an operator mistakenly selects invalid voltage or current settings, your application should handle it appropriately. Institute as many safeguards

as possible to prevent problems. Select a lifecycle development model that helps you find problems as early as possible, and allow time for formal reviews and thorough testing.

Configuration Management

Configuration management is the process of controlling changes and ensuring that they are reviewed before they are made. In Chapter 2, *Development Models*, development models such as the waterfall model are discussed. A central focus of these models is to convert software development from a chaotic unplanned activity to a controlled process. These models improve software development by establishing specific, measurable goals at each stage of development.

Regardless of how well development proceeds, changes will need to be implemented. Customers might introduce new requirements in the design stage. Performance problems discovered during development might prompt reevaluation of the design. You might need to rewrite a section of code to correct a problem found in testing. Changes can affect any components of the project from the requirements and specification to the design, code, and tests. If these changes are not made carefully, you might introduce problems that can delay development or degrade quality.

Source Code Control

After setting the project quality requirements, develop a process to handle changes. This is very important for projects with multiple developers. As the developers work on VIs, they need a method for collecting and sharing their work. A simple method to handle this is to establish a central source repository. If each of the developers' computers is networked, you can create a shared location that serves as a central source for development. When they need to modify files, they can retrieve them from this location. When they are finished with the changes and the system is working, they can return the files to this location.

Common files and areas of overlap introduce the potential for accidental loss of work. If two developers decide to work on the same VI at the same time, only one developer really can merge changes into the project. The other developer's efforts will be lost. You might avoid this with good communication, if each developer notifies the others

when he or she needs to work on a specific VI. Inevitably, however, a mistake will be made, and work will be lost.

Source Code Control tools address the problems of sharing VIs and controlling access to avoid accidental loss of data. Source Code Control tools make it easy to set up shared projects and to retrieve the latest files from the server. Once you have created a project, you can check out a file for development. Checking out a file marks it with your name so that no other developer can modify the file. Other developers can, however, retrieve the current version from the server. A developer can check out the file, make modifications, test the changes, and check in the file to the source code system. After the file is checked in, it is accessible to the whole development team. Then another developer can check out the file to make further modifications.

The G Source Code Control tools are accessible from the **Project** menu. They are described in detail in Chapter 11, *Source Code Control Tools*.

Managing All Project-Related Files

The G Source Code Control tools can manage more than just VIs. You can use them to manage all aspects of your project. Requirements, specifications, illustrations, reviews, and other documents related to your project all can be managed within the Source Code Control system. This ensures that you can control access to these documents and share them as needed. You can use the tools to track changes and access older versions of files.

As described in Chapter 4, *Prototyping and Design Techniques*, source management of all project-related files is extremely important for developing quality software. Source management is a requirement for certification under one of the existing quality standards such as ISO 9000.

Retrieving Old Versions of Files

There are times when you need to retrieve an old version of a file or project. This might happen if you make a change to a file and check it in, only to realize that you made a mistake. Another reason it might happen is if you send a beta version of your software to a customer and continue development. If the customer reports a problem, you might need to access a copy of the beta version.

One way to achieve this is to back up your files periodically. However, unless you back up your system after every change, you might not have access to every version.

The G Source Code Control tools provide a way to check in new versions of a file and make a back-up copy of the old version. Depending on how you configure the system, the tools can maintain multiple backup copies of a file.

You can use the tools to label versions of files with descriptive names like *beta*, *v1.0*, and so on. You can label any number of files and later retrieve all versions of a file with a specific label. When you release a version of your software, take a snapshot of the files by attaching a label to them. Chapter 11, *Source Code Control Tools,* describes the file and system history options.

Tracking Changes

If you are managing a software project, it is important to monitor changes and track progress toward specific milestone objectives. You also can use this information to determine problem areas of a project by identifying which components required a lot of changes.

The G Source Code Control tools maintain a log of all changes made to files and projects. When checking in a file, the developer is prompted to enter a summary of the changes made. This summary information is added to that file's log.

You can view the history information for a file or for the system and generate reports containing that information. See the *SCC File History*, *System History*, and *Creating Reports* sections of Chapter 11, *Source Code Control Tools*, for more information.

Change Control

Large projects might require a formal process for evaluation and approval of each change request. A formal evaluation system like this might be too restrictive, so be selective when choosing the control mechanisms that you introduce into your system.

Changes to specific components, such as documents related to user requirements, must be handled cautiously, because they generally are worked out through several iterations with the customer. In this case, *customer* is used in a general sense; you might be your own customer, other departments in your company might be your target audience, or

you might develop the software under contract for a third party. When you are your own customer, it is much easier to adjust requirements as you move through the specification and even the design stage. If you are developing for someone else, changing requirements can be extremely difficult.

Source Code Control tools give you a degree of control. You have traceability for all changes, and you can configure the system to maintain previous versions so that you can back out of changes if necessary. Some Source Code Control systems give you more options for controlling software change. For example, with Microsoft Visual SourceSafe, you can control access to files, so that some users have access to specific files while others do not. Or, you can specify that files can be retrieved by anyone, but only certain users can make modifications.

With this kind of control access, you might limit change privileges for requirement documents to specific team members. Or, you might control access so that a user has modify access privileges to a file only when the change request is approved.

The amount of control you apply can vary throughout the development process. In the early stages of the project, before formal evaluation of the requirements, you do not necessarily need to restrict change access to files, nor do you need to follow formal change request processes. Once the requirements are approved, however, you can institute stronger controls. You can apply the same concept of varying the level of control before and after a project phase is complete to specifications, test plans, and code as well.

Testing Guidelines

You should decide up front what level of testing is expected. Engineers under deadline pressure frequently give short attention to testing, devoting more time to other development. Most software engineers will tell you, however, that a certain level of testing is guaranteed to save you time in the end.

The degree to which you expect developers to test should be clearly understood. Also, testing methodologies should be standardized, and results of tests should be tracked. As you develop the requirements and design specifications, you also should develop a test plan to help you verify that the system and all of its components work. Testing should

reflect the quality goals you want to achieve. For example, if performance is more critical than robustness, you should develop more tests for performance, and fewer that attempt incorrect input, low-memory situations and the like.

Testing should not be an afterthought. It should be considered part of the initial design phases and should be implemented throughout development to find and fix problems as soon as possible.

There are a variety of testing methodologies you can use to help increase the quality of your VI projects. The following sections describe some testing methodologies.

Black Box and White Box Testing

Black box testing is based on the expected functionality of software, without knowledge of how it works. It is called black box testing because you cannot see the internal workings. Black box testing can be done based largely on a knowledge of the requirements and the interface of a module. For a subVI, you could perform black box tests on the interface of a subVI to evaluate results for various input values. If robustness is a quality goal, you should include erroneous input data to see if the subVI handles it well. For example, for numeric inputs, you should see how the subVI handles Infinity (Inf, -Inf), Not a Number (NaN), and other out-of-range values. See the *Unit Testing* section for more examples.

White box testing is designed with knowledge of the internal workings of the software. Use white box testing to check that all of the major paths of execution are exercised. By examining a block diagram and looking at the conditions of case structures and the values controlling loops, you can design tests that check those paths. White box testing on a large scale is impractical because it is difficult to test all possible paths.

While white box testing is difficult to fully implement for large programs, you can choose to test the most important or complex paths. White box testing can be combined with black box testing for more thorough testing of software.

Unit, Integration, and System Testing

Black box and white box testing can be used to test any component of software, regardless of whether it is an individual VI or the complete application. Unit, integration, and system testing are phases of your project at which you can apply black box and white box tests.

Unit Testing

Unit testing concentrates on testing individual software components. For example, you might test an individual VI to see that it works correctly, handles out-of-range data, has acceptable performance, and that all major execution paths in its block diagram are executed and performed correctly. Individual developers can perform unit tests as they work on their modules.

Some examples of common problems unit tests might account for are listed below:

- Boundary conditions for each input, such as empty arrays and empty strings, or 0 for a size input. Be sure that floating point parameters handle Infinity and Not a Number.
- Invalid values for each input, such as –3 for a size input.
- Strange combinations of inputs.
- Missing files and bad path names.
- What happens when the user clicks the CANCEL button in a file dialog box?
- What happens if the user aborts the VI?

Define various sets of inputs that thoroughly test your VI, then write a test VI that calls your VI with each combination of inputs and checks the results. Or, use interactive data logging to create your input sets (test vectors), and replay them interactively to re-test the VI or automatically from a test VI that uses programmatic data retrieval.

To perform unit testing, you might need to *stub out* some components that have not been implemented yet or that are being developed. For example, if you are developing a VI that communicates with an instrument and writes information to a file, another developer can work on a file I/O driver that writes out the information in a specific format. To test your components early, you might choose to stub out the file I/O driver by creating a VI with the same interface. This VI can write the data out in a format that is easy for you to check. You can test

the driver with the real file I/O driver later during the integration phase as described in the next section.

Regardless of how you test your VIs, record exactly how, when, and what you tested and keep any test VIs you created. This test documentation is especially important if you are creating VIs for paying customers; but it is also useful just for yourself. When you revise your VIs, you should run the existing tests to make sure you have not broken anything. You also should update the tests for any new functionality you have added.

Integration Testing

Integration testing is performed on a combination of units. While unit testing finds most bugs, integration testing might reveal unanticipated problems. Modules might not work together as expected. They might interact in unexpected ways because of the way they manipulate shared data. For more information, see Chapter 26, *Performance Issues*, in the *G Programming Reference Manual*, or Chapter 27, *Performance Issues*, in the *LabVIEW User Manual*.

Integration testing also can be done in earlier stages before you put the whole system together. For example, if a developer creates a set of VIs that communicates with an instrument, he or she could develop unit tests that verify that each subVI correctly sends the appropriate commands. He or she also could develop integration tests that use several of the subVIs in conjunction with each other, to verify that there is not any unexpected interaction.

Integration testing should not be performed as a comprehensive test in which you combine all the components and try to test the top-level program. Doing this can be very expensive because it is difficult to determine the specific source of problems within a large set of VIs. Instead, you should consider testing incrementally, either with a top-down or bottom-up testing approach.

With a top-down approach, you gradually integrate major components, testing the system with the lower-level components of the system disabled—or stubbed out—as described in the *Unit Testing* section. Once you have verified that the existing components work together within the existing framework, you can enable additional components.

With a bottom-up approach, you test low-level modules first and gradually work up toward the high-level modules. Begin by testing a small number of components combined into a simple system, such as the driver test described above. After you have combined a set of modules and verified that they work together, add components and test them with the already-debugged subsystem.

The bottom-up approach consists of tests that gradually increase their scope, while the top-down approach consists of tests that are gradually refined as new components are added.

Regardless of the approach you take, you must perform regression testing at each step to verify that the features that already have been tested still work. Regression testing consists of repeating some or all previous tests. Because you might need to perform the same tests numerous times, you might want to develop representative subsets of tests that can be used for frequent regression tests. These components can be run at each stage, while the more detailed tests can be run to test an individual set of modules if problems are encountered or as part of a more detailed regression test that is applied periodically during development.

System Testing

System testing happens after integration to determine whether the product meets customer expectations and to make sure that the software works as expected within the hardware system. This can be done first as a set of black box tests that verify that the requirements have been met. Most LabVIEW and BridgeVIEW applications perform some kind of I/O. The application also might communicate with other applications. With system testing, you test the software to make sure that it fits into the overall system as expected. When testing the system, you will ask and answer questions such as the following:

- Are performance requirements met?
- If my application communicates with another application, does it handle an unexpected failure of that application well?

You can complete this testing with alpha and beta testing. Alpha and beta testing serve to catch test cases that might not have been considered or completed by the developers. With alpha testing, a functionally complete product is tested in-house to see if any problems are found. When alpha testing is complete, the product is beta tested by customers in the field.

Alpha and beta testing are the only testing mechanisms for some companies. This is unfortunate because alpha and beta testing actually can be inexact. Alpha and beta testing are not a substitute for other forms of testing that rigorously test each component to verify that it meets stated objectives. Because this type of testing is done late in the development process, it is difficult and costly to incorporate changes suggested as a result.

Formal Methods of Verification

Some software engineers are proponents of formal verification of software. While other testing methodologies attempt to find problems by exploration, formal methods attempt to *prove* the correctness of software mathematically. If you have ever worked through proofs in math classes, you have an idea of what is involved.

The principal idea is to analyze each function of a program to determine if it does what you expect. You mathematically state the list of preconditions before the function and the postconditions that are present as a result of the function. This process can be performed either by starting at the beginning of the program and adding conditions as you work through each function or by starting at the end and working backward, developing a set of weakest preconditions for each function. This process is described briefly in some of the documents listed in Appendix A, *References*.

This type of testing becomes more complex as more and more possible paths of execution are added to a program through the use of loops and conditions. Many people believe that formal testing presents interesting ideas for looking at software that can help in small cases, but that it is impractical for most programs.

Style Guidelines

Inconsistent approaches to development and to user interfaces can be a problem when multiple developers work on a project. Each developer has his or her own style of development, color preferences, display techniques, documentation practices, and diagram methodologies. One developer might make extensive use of global variables and sequence structures while another might prefer to make more use of data flow.

Inconsistent style techniques can create software that, at a minimum, looks bad. If the user interface VIs have different behaviors— some expect a user to click a button when he or she is finished and others expect a keyboard function key—users might become confused and find the VIs difficult to use.

Inconsistent style also makes software difficult to maintain. For example, if one developer does not like to use subVIs and decides to develop all features within a single large VI, it will be difficult to modify.

Establish a set of guidelines for your VI development team. Establish an initial set of guidelines and add additional rules as the project progresses. These style guidelines can be used in future projects.

Chapter 7, *Using Consistent Style—The G Style Guide*, provides some style recommendations. Use these guidelines as a basis for developing your own style guide. Programming style in any language is something for which there really cannot be a single standard. What one group prefers, another group might disagree with. Select a set of guidelines that works for you and your development team.

Design Reviews

Design reviews are a great way to identify and fix problems during development. When the design of a feature is complete, set up a design review with at least one other developer. Discuss quality goals, asking questions such as the following.

- Does the design incorporate testing?
- Is error handling built-in?
- Are there any assumptions in the system that might be invalid?

Also, look at the design with an eye for features that are essential as opposed to features that are extras. While there is nothing wrong with building in extra features, if quality and schedule are important, you should ensure that these extra features are either scheduled for late in the development process, so that they can be dropped, or moved to the list of features for subsequent releases. Be sure to document the results of the design review and any changes that are recommended.

Code Walkthroughs

A *code walkthrough* is similar to a design review, except it analyzes the code instead of the design. To perform a code review, give printouts of the VIs to review to one or more developers. You might want to perform the review online because VIs are easier to read and navigate online. The designer should talk through the design. The reviewers compare the description to the actual implementation. The reviewers should consider many of the same issues included in a design review. During a code walkthrough, many of the following questions might be asked and answered.

- What happens if an error is returned by a specific VI or function? Are errors handled and/or reported correctly?
- Are there any race conditions? An example of a race condition is a block diagram that reads from and writes to a global variable, and there is the potential that a parallel block diagram could simultaneously attempt to manipulate the same global variable, resulting in loss of data.
- Is the block diagram well-implemented? Are the algorithms efficient in terms of speed and/or memory usage? For more information, refer to Chapter 26, *Performance Issues*, in the *G Programming Reference Manual*, or Chapter 27, *Performance Issues*, in the *LabVIEW User Manual*.
- Is the block diagram easy to maintain? Has the developer made good use of hierarchy, or is he or she placing too much functionality in a single VI? Are the group's style guidelines adhered to?

There are a number of other features you can look for in a code walkthrough. Take notes on the problems you encounter and add them to a list you can use as a guideline for other walkthroughs.

Stick to technical issues when doing a code walkthrough. Remember to review only the code, not the developer who produced it. Try not to focus on the negative; be sure to raise positive points.

For documentation that contains additional ideas on walkthrough techniques, see Appendix A, *References*.

Postmortem Evaluation

At the end of each stage in the development process, you should consider having a *postmortem meeting* to discuss what has gone well and what has not. Each developer should evaluate the project and be frank in discussing obstacles that reduce the quality level of the project. Each developer should consider the following questions.

- What are we doing right? What is working well?
- What are we doing wrong? What can be improved?
- Are there specific areas of the design/code that need a lot of work? Is a design review or code walkthrough of that section necessary?
- Are the quality systems working? Could we catch more problems if we changed the quality requirements? Are there better ways to get the same results?

Postmortem meetings at major milestones can help to correct problems mid-schedule instead of waiting until the release is complete.

Software Quality Standards

As software has become a more critical component in systems, concerns about software quality have increased. Consequently, a number of organizations have developed quality standards that either are specific to software or can be applied to software. When developing software for some large organizations, especially government organizations, you might be required to follow one of these standards.

The following sections include a brief overview of the most popular standards. Appendix A, *References*, lists several documents that contain more information on these standards.

International Organization for Standards (ISO) 9000

The International Organization for Standards has developed the ISO 9000 family of standards for quality management and assurance. Many countries have adopted these standards. In some cases, governmental bodies require compliance with this ISO standard. Compliance generally is measured by certification performed by a third-party auditor. The ISO 9000 family is widely used within Europe and Asia. It has not been widely adopted within the United States,

although many companies and some government agencies are starting to adopt it.

In each country, the ISO family of standards might be referred to by slightly different names. For example, in the United States it has been adopted as the ANSI/American Society for Quality Control (ASQC) Q90 Series. In Europe, it has been adopted by the European Committee for Standardization (CEN) and the European Committee for Electrotechnical Standardization (CENELEC) as the European Norm (EN) 29000 Series. In Canada, it has been adopted by the Canadian Standards Association (CSA) as the Q 9000 series. But it is most commonly referred to as ISO 9000.

ISO 9000 is an introduction to the ISO 9000 family of standards. ISO 9001 is a model for quality assurance in design, development, production, installation, and servicing. Its focus on design and development makes it the most appropriate for software products.

Because the ISO 9000 family is designed to apply to any industry, it can be somewhat difficult to apply to software development. ISO 9000.3 is a set of guidelines designed to explain how to apply ISO 9001 to software development.

ISO 9001 does not dictate software development procedures. Instead, it requires documentation of development procedures and adherence to the standards you set. Conformance with ISO 9001 does not guarantee quality. Instead, the idea behind ISO 9001 is that companies that emphasize quality and follow their documented practices will produce higher quality products than companies that do not.

U.S. Food & Drug Administration (FDA) Standards

The FDA is in the process of finalizing new rules for software used in medical applications. The FDA requires all software used in medical applications to meet its Current Good Manufacturing Practices (CGMP). One of the goals of the new standard is to make it as consistent as possible with ISO 9001 and a supplement to ISO 9001, ISO/CD 13485. While these FDA standards are largely consistent with ISO 9001, there are some differences. Specifically, the FDA did not feel that ISO 9001 was specific enough about certain requirements, so they have spelled them out in the new rules.

Details of the new CGMP rules and how they compare to ISO 9001 can be found at the FDA internet home page at `http://www.fda.gov`.

Capability Maturity Model (CMM)

In 1984, the United States Department of Defense created the Software Engineering Institute (SEI) to establish standards for software quality. The SEI developed a model for software quality called the Capability Maturity Model (CMM). The CMM focuses on improving the maturity of an organization's processes.

Whereas ISO establishes only two levels of conformance (pass or fail), the CMM ranks an organization into one of five categories:

Level 1. Initial—The organization has few defined processes; quality and schedules are unpredictable.

Level 2. Repeatable—Groups establish policies based on software engineering techniques and previous projects that allow repeated success. Groups use configuration management tools to manage projects. Also, they track software costs, features, and schedules. Project standards are defined and followed. While the groups can handle similar projects based on this experience, their processes might not be mature enough to handle significantly different types of projects.

Level 3. Defined—The organization establishes a baseline set of policies for all projects. Groups are well-trained and know how to customize this set of policies for specific projects. Each project has well-defined characteristics that make it possible to accurately measure progress.

Level 4. Managed—The organization sets quality goals for projects and processes and measures progress toward those goals.

Level 5. Optimizing—The organization emphasizes continuous process improvement across all projects. The organization evaluates the software engineering techniques it uses in different groups and applies them throughout the organization.

Figure 3-1 illustrates the five levels of the CMM and the processes necessary for advancement to the next level.

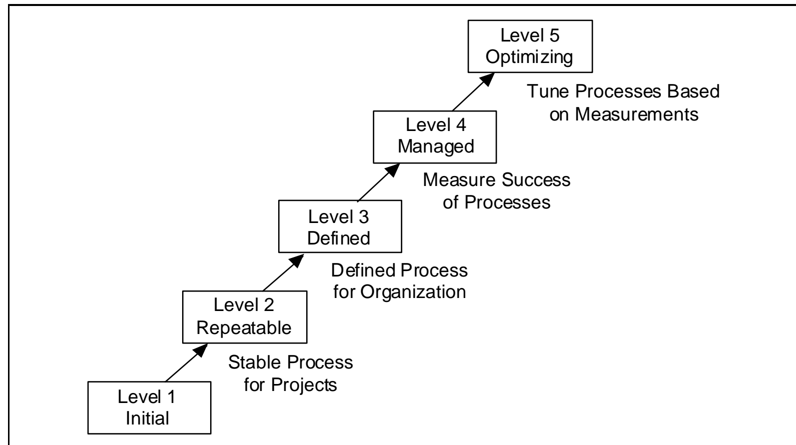


Figure 3-1. Capability Maturity Model

Most companies are at level one or two. The U.S. Department of Defense prefers a level three or higher CMM assessment in bids on new government software development. The CMM also is used in some commercial companies, mainly in the United States.

The CMM differs from ISO 9001 in that it is software-specific. Also, the ISO specifications are fairly high-level (ISO 9001 is only a few pages) while CMM is very detailed (more than 500 pages).

Institute of Electrical and Electronic Engineers (IEEE) Standards

IEEE has defined a number of standards for software engineering. IEEE Std 730, first published in 1980, is a standard for software quality assurance plans. This standard serves as a foundation for several other IEEE standards, and gives a brief description of the minimum requirements for a quality plan in the following areas:

- Purpose
- Reference documents
- Management
- Documentation
- Standards, practices, conventions, and metrics
- Reviews and audits

- Test
- Problem reporting and corrective action
- Tools, techniques, and methodologies
- Code control
- Media control
- Supplier control
- Records collection, maintenance, and retention
- Training
- Risk Management

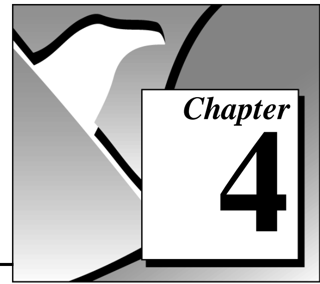
As with the ISO standards, IEEE Std 730 is fairly short. It does not dictate how you meet the requirements but does require documentation for these practices to a specified minimum level of detail.

In addition to IEEE Std 730 there are several other IEEE standards related to software engineering, including:

- Std 610—Defines standard software engineering terminology
- Std 829—Establishes standards for software test documentation
- Std 830—Explains the content of good software requirements specifications
- Std 1074—Describes the activities that should be performed as part of a software lifecycle without requiring a specific lifecycle model
- Std 1298—Details the components of a software quality management system; similar to ISO 9001

Your projects might be required to meet some or all of these standards. Even if you are not required to develop to one of these specifications, they can be helpful in developing your own requirements, specifications, and quality plans.

Prototyping and Design Techniques



This chapter gives you pointers for project design, including programming approaches, prototyping, and benchmarking.

When you first begin a programming project, deciding how to start can be intimidating. A lot of G programmers start immediately with a code and fix development process, building up some of the VIs they think they will need. Then they realize that they actually need something different from what they have built already. Consequently, a lot of code is developed, reworked, or thrown away unnecessarily.

Clearly Define the Requirements of Your Application

Before developing a detailed design of your system, you should define your goals as clearly as possible. Begin by making a list of requirements. Some requirements are very specific, such as the types of I/O, sampling rates, or the need for real-time analysis. You need to do some research at this early stage to be sure you can meet the specifications. Other requirements depend on user preference, such as file formats or graph styles.

Try to distinguish between absolute requirements and desires. You might be able to satisfy all requests, but it is best to have an idea about what you can sacrifice if you run out of time.

Also, be careful that the requirements are not so detailed that they constrain the design. For example, in designing an I/O system the customer probably has certain sampling rate and precision requirements. He or she also is constrained by cost. Those issues should be included in your requirements. However, if you can avoid specifying the operating system and hardware, you can adjust your design after you begin prototyping and benchmarking various components. As long as the costs are within budget, and the timing and precision issues are met, the customer might not care whether the system uses a particular type of plug-in card or other hardware.

Another example of overly constraining a design is to be too specific about the format for display used in various screens with which the customer interacts. A picture of a display might be useful to explain requirements, but be clear about whether the picture is a requirement or a guideline. Some designers go through significant contortions trying to produce a system that behaves in a specific way because a certain behavior was a requirement. In this case, there might be a simpler solution that produces the same results at a much lower cost in a shorter time.

Top-Down Design

The block diagram programming metaphor used in G was designed to be easy to understand. Most engineers already use block diagrams to describe systems. The goal of the block diagram is to make it easier for you to move from the system block diagrams you create to executable code.

The basic concept is to divide the task into manageable pieces at logical places. Begin with a high-level block diagram that describes the main components of your system. For example, you might have a block diagram that consists of a block for configuration, a block for acquisition, a block for analysis of the acquired data, a block for saving the data to disk, and a block to clean up at the end of the system.

After you have determined the high-level blocks, create a block diagram that uses those blocks. For each block, create a new *stub* VI (a non-functional prototype representing a future subVI). Create an icon for this stub VI and create a front panel with the necessary inputs and outputs. You do not have to create a block diagram for this VI yet. Instead, define the interface and see if this stub VI is a useful part of your top-level block diagram.

After you assemble a group of these stub VIs, determine the function of each block and how it works. Ask yourself whether any given block generates information that some subsequent VI needs. If so, make sure that your top-level block diagram sketch contains wires to pass the data between the VIs. You can document the functionality of the VI and the inputs and outputs using the Info VI and Description tools in LabVIEW and BridgeVIEW.

In analyzing the transfer of data from one block to another, try to avoid *global variables*, because they hide the data dependency between VIs and might introduce race conditions. See Chapter 26, *Performance Issues*, of the *G Programming Reference Manual*, or Chapter 27, *Performance Issues*, of the *LabVIEW User Manual*, for more information. As your system becomes larger, it becomes difficult to debug if you use global variables as your method of transferring information between VIs.

Continue to refine your design by breaking down each of the component blocks into more detailed outlines. You can do this by going to the block diagram of what was once a stub VI and filling out its block diagram, placing lower-level stub VIs on the block diagram that represent each of the major actions the VI must perform.

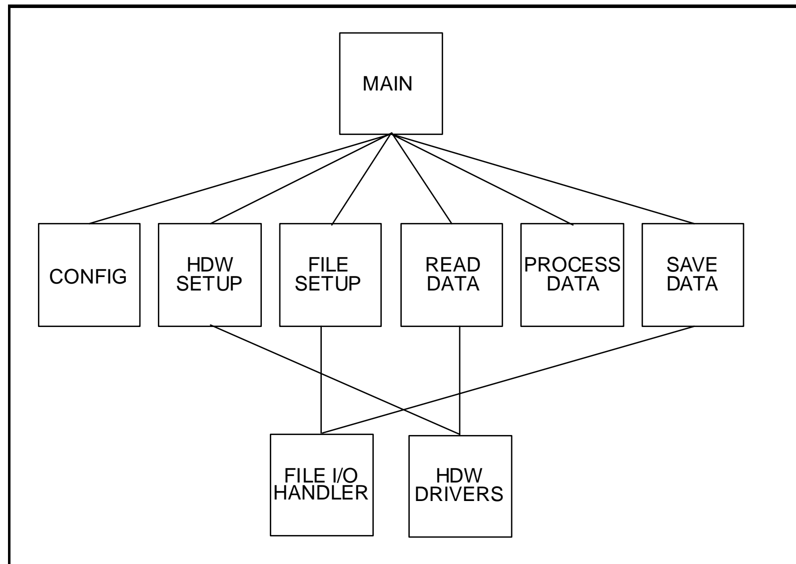
Be careful not to jump too quickly into implementing the system at this point. One of the objectives here is to gradually refine your design so you can determine whether you have left out any necessary components at higher levels. For example, when refining the acquisition phase, you might realize that there is more information you need from the configuration phase. If you completely implement one block before you analyze a subsequent block, you might need to redesign the first block significantly. It is better to try to refine the system gradually on several fronts, with particular attention to sections that have more risk because of their complexity.

The following example illustrates how you might apply top-down design techniques to a data acquisition system.

Example—A Data Acquisition System

This example describes how you might design a general data acquisition system. This system must let the user provide some configuration of the acquisition (rates, channels, and so on), acquire data, process the data, and save it to disk.

Start to design the VI hierarchy by breaking the problem into logical pieces. As shown in the following flowchart, there are several major blocks that you can expect to see in one form or another on every data acquisition system.



Think about the data structures that you will need, asking questions such as “What information needs to accompany the raw data values from the Read Data VI to the Save Data VI?” This might imply a cluster array—an array of many channels, each element of which is a cluster containing the value, the channel name, scale factors, etc. A method that performs some action on such a data structure is called an *algorithm*. Algorithms and data structures are closely intertwined. This is reflected in modern structured programming, and it works well in G. If you like to use pseudo-code, try that technique as well. The following figures show a relationship between pseudo-code and G structures.

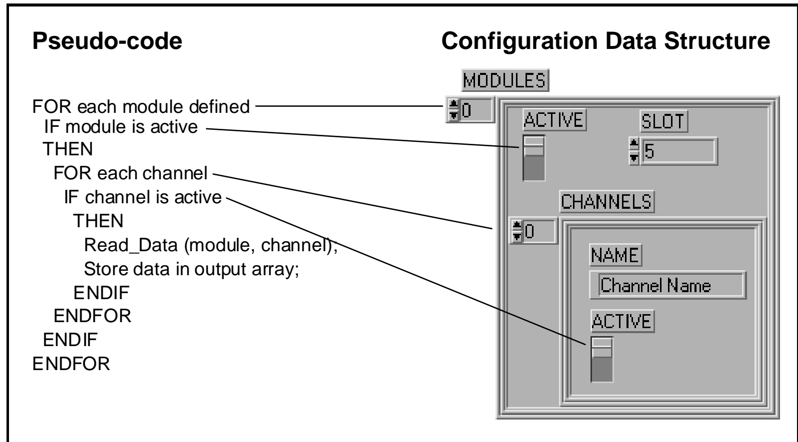


Figure 4-1. Mapping Pseudo-Code into a G Data Structure

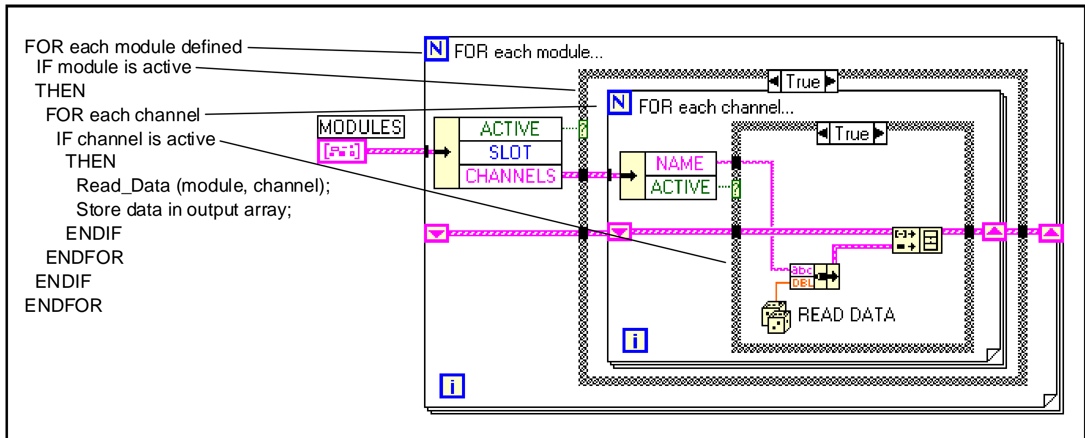


Figure 4-2. Mapping Pseudo-Code into Actual G Code

Notice the correspondence between the program and the data structure.

Many experienced LabVIEW and BridgeVIEW users prefer to use G sketches. You can draw caricatures of the familiar structures and wire them together on paper. This is a good way to think things through,

sometimes with the help of other G programmers. If you are not sure how a certain function will work, prototype it in a simple test VI.

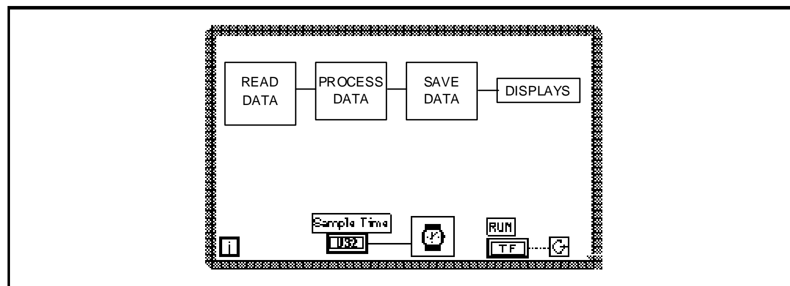


Figure 4-3. Data Flow for a Generic Data Acquisition Program

Artificial data dependency between the initialization VIs and the main While structure eliminates the need for a Sequence structure. Finally, you are ready to write the program in G. Remember to make your code modular, building subVIs when there is a logical division of labor or the potential for code reuse. Solve the more general problems along with your specific ones. Test your subVIs as you write them. This might involve construction of higher-level test routines. It is much easier to catch the bugs in one small module than in a large hierarchy of VIs.

Bottom-Up Design

Usually, you should avoid bottom-up system design, although it is sometimes useful when used in conjunction with top-down design. Bottom-up design is the exact opposite of top-down design. You start by building the lower-level components and then progress up the hierarchy, gradually putting pieces together until you have the complete system.

Because you do not start with a clear idea of the big picture, the problem with bottom-up design is that you might build pieces that do not fit together the way you expect.

There are specific cases in which using bottom-up design is appropriate. If the design is constrained by low-level functionality, you might need to build that low-level functionality first to get an idea of how it can be used. This might be true of an instrument driver, where the command set for the instrument constrains you in terms of when you can do certain operations. For example, with a top-down design, you might break up your design so that configuration of the instrument and reading

a measurement from the instrument are done in distinct VIs. The instrument command set might turn out to be more constraining than you thought, requiring you to combine these operations. In this case, with a bottom-up strategy, you might start by building VIs that deal with the instrument command set.

In most cases, you should use a top-down design strategy. You might mix in some components of bottom-up design, if necessary. Thus, in the case of an instrument driver, you might use a risk-minimization strategy to understand the limitations of the instrument command set and develop the lower-level components. Then use a top-down approach to develop the high-level blocks.

The following example shows in more detail how you can apply this technique to the process of designing a driver for a GPIB instrument.

Example—An Instrument Driver

A complex GPIB-controlled instrument can have hundreds of commands, many of which interact with each other. A bottom-up approach might be the most effective in designing a driver for such an instrument. The key here is that the problem is detail-driven—you must learn the command set and design a front panel that is simple for the user, yet gives full control of the instrument functionality. Design a preliminary VI hierarchy, preferably one based on similar instrument drivers. You must satisfy the user's needs. Designing a driver requires more than putting knobs on GPIB commands. The example chosen here is the Tektronix 370A Curve Tracer. It has about 100 GPIB commands if you include the read and write versions of each one.

Once you begin programming, the hierarchy will fill out naturally, one subVI at a time. Add lower-level support VIs as required, such as: a communications handler, a routine to parse a complex header message, or an error handler. For instance, the 370A required a complicated parser for the waveform preamble that contains such information as scale factors, offsets, sources and units. It was much cleaner to bury this operation in a subVI than to let it obscure the function of a higher level VI. Also, a communications handler made it very simple to exchange messages with the instrument. Such a handler formats and sends the message, reads the response (if required), and checks for errors.

Once the basic functions are ready, assemble them into a demonstration driver VI that makes the instrument do something useful. You will

quickly find any fundamental flaws in your earlier choices of data structures, terminal assignments, and default values.

The *LabVIEW Instrument I/O Reference Manual* describes this development process in detail.

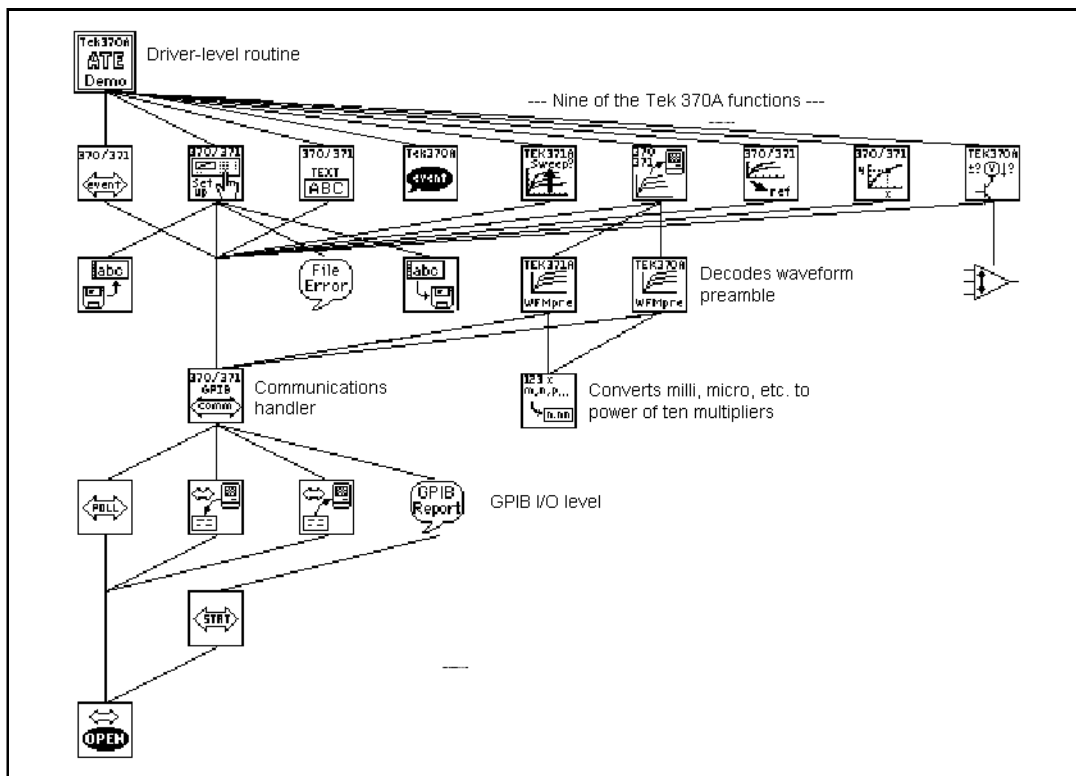


Figure 4-4. VI Hierarchy for the Tektronix 370A

The top-level VI is an automated test example. It calls nine of the major functions included in the driver package. Each function, in turn, calls subVIs to perform GPIB I/O, file I/O, or data conversion.

Designing for Multiple Developers

One of the main challenges in the planning stage is to establish discrete project areas for each developer. As you design the specification and architectural design, you should begin to see areas that have a minimal amount of overlap. For example, a complicated data monitoring system might have one set of VIs that displays and manipulates data and another set that acquires the information and transfers it to disk. These two modules are substantial, do not overlap, and can be assigned to different developers.

Inevitably, there will be some interaction between the modules. One of the principal objectives of the early design work is to design how those modules interact with each other. The data display system must access the data it needs to display. The acquisition component needs to provide this information for the other module. At an early stage in development, you might design the connector panes of VIs needed to transfer information between the two modules. Likewise, if there are global data structures that must be shared, these should be analyzed and defined early in the architectural design stage before the individual developers begin work on their components.

In the early stages, each developer can create stub VIs with the connector pane interface that was defined for the shared module. This stub VI can do nothing, or if it is a VI that returns information, you could have it generate random data. This allows each member of the development team to continue development without having to wait for the other modules to be finished. It also makes it easy for the individuals to perform unit testing of their modules as described in Chapter 3, *Incorporating Quality into the Development Process*.

As components near completion, you can integrate the modules by replacing the stub components with their real counterparts. At this point you can perform integration testing (see the *Integration Testing* section in Chapter 3, *Incorporating Quality into the Development Process*) to verify the system works as a whole.

Front Panel Prototyping

As mentioned in the Chapter 2, *Development Models*, front panel prototypes can provide insight into the organization of your program. Assuming your program is user interface–intensive, you can attempt to mock up an interface that represents what the user sees.

Avoid implementing block diagrams in the early stages of creating prototypes so you do not fall into the code and fix trap. Instead, create just the front panels, and as you create buttons, list boxes, and rings, think about what should happen as the user makes selections, asking questions such as the following.

- Should the button lead to another front panel?
- Should some controls on the front panel be hidden and replaced by others?

If new options are presented, follow those ideas by creating new front panels to illustrate the results. This kind of prototyping can help solidify the requirements for a project and give you a better idea of its scope.

Prototyping cannot solve all development problems, however. You have to be careful how you present the prototype to customers. Prototypes can give an overly inflated sense that you are rapidly making progress on the project. You have to be clear to the customer, whether it is an external customer or other members of your company, that this prototype is strictly for design purposes and that much of it will be reworked in the development phase.

Another danger in prototyping is that you might overdo it. Consider setting strict time goals for the amount of time you will prototype a system to prevent yourself from falling into the code and fix trap.

Of course, front panel prototyping only deals with user interface components. As described here, it does not deal with I/O constraints, data types, or algorithm issues in your design. The front panel issues might help you to better define some of these areas, because it gives you an idea of some of the major data structures you need to maintain, but it does not address all of these issues. For those, you need to use one of the other methods described in this chapter.

Performance Benchmarking

For I/O systems with a number of data points or high transfer rate requirements, test the performance-related components early, because the test might prove your design assumptions are incorrect.

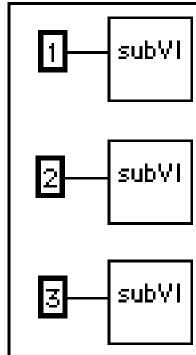
For example, if you plan to use an instrument as your data acquisition system, you might want to build some simple tests that perform the type of I/O you plan to use. While the specifications might seem to indicate that the instrument can handle the application you are creating, you might find that triggering, for example, takes longer than you expected, or that switching between channels with different gains cannot be done at the necessary rate without reducing the accuracy of the sampling. Or perhaps even though the instrument can handle the rates, you do not have enough time on the software side to perform the desired analysis.

A simple prototype of the time-critical sections of your application can help to reveal this kind of problem. The timing template example in the `examples/general/timing` directory illustrates how to time a process. Because timings can fluctuate from one run to another for a variety of reasons (initial run might take longer because it allocates buffers, system interrupts, screen updates and user interaction can cause it to take longer in some cases), you should put the operation in a loop and display the average execution time. You also can use a graph to display timing fluctuations.

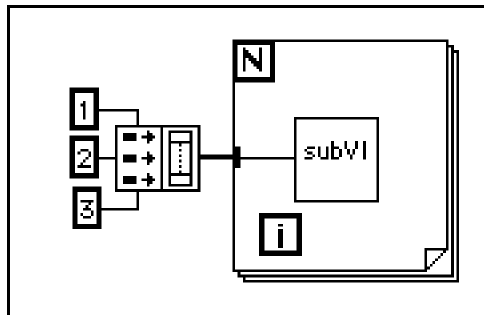
Identify Common Operations

As you design your programs, you might find that certain operations are performed frequently. Depending on the situation, this might be a good place to use subVIs or loops to repeat an action.

For example, consider the following figure, where three similar operations run independently.



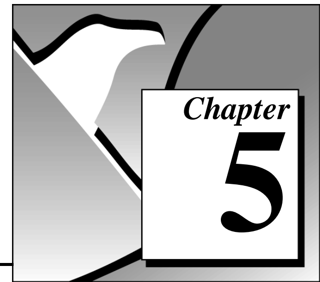
An alternative to this design is a loop that performs the operation three times. You can build an array of the different arguments and use auto-indexing to set the correct value for each iteration of the loop.



If the array elements are constant, you can use an array constant instead of building the array on the block diagram.

Some users mistakenly avoid using subVIs because they are afraid of the overhead it might add to their execution time. It is true that you probably do not want to create a subVI from a simple mathematical operation such as the **Add** function, especially if it must be repeated thousands of times. However, the overhead for a subVI is fairly small, and usually is dwarfed by any I/O you perform or by any memory management that might occur from complex manipulation of arrays.

Scheduling and Project Tracking



This chapter describes techniques for developing estimates of development time and using those estimates to create schedules. This chapter also distinguishes between an estimate, which reflects the time required to implement a feature, and a schedule, which reflects how you fulfill that feature. Estimates are commonly expressed in ideal person-days (eight hours of work). In creating a schedule from estimates, you must consider dependencies (one project might have to be completed before another can begin) and other tasks (meetings, support for existing projects, and so on).

Estimation

One of the principle tasks of planning is to estimate the size of the project and fit it into the schedule. Most projects are at least partially schedule-driven. Schedule, resources, and critical requirements all interact to determine what you can implement in a release.

Unfortunately, when it comes to estimating software schedules accurately, very few people are successful. Major companies all have had software projects exceed original estimates by a year or more. Poor planning or an incomplete idea of project goals often causes deadlines to be missed. Another major cause of missed schedules is known as *feature creep*—your design gradually grows to include features that were not part of the original requirements. In many cases, the slips in schedule are due to the use of a code and fix development process rather than a more measurable development model.

Off-the-cuff estimates are almost never accurate for the following reasons.

- People are usually overly optimistic. An estimate of two months at first might seem like an infinite amount of time. Then, during the last two weeks of the project, when developers find themselves working many overtime hours, it becomes clear that it is not.
- The objectives, implementation issues, and quality requirements are not understood clearly. When challenged with the task of

creating a data monitoring system, an engineer might estimate two weeks. If the product is designed by the engineer and for the engineer, this estimate might be right. However, if it is for other users, he or she probably is not considering requirements that might be assumed by a less knowledgeable user but never are specified clearly.

For example, VIs need to be reliable and easy-to-use because the engineer is not going to be there to correct them if a problem occurs. A considerable amount of testing and documentation is necessary. Also, the user needs to save results to disk, print reports, and view and manipulate the data on screen. If he or she has not discussed or considered the project in detail, the engineer is setting himself or herself up for failure.

- Day-to-day tasks are ignored. There are meetings to attend, holidays, reports to write, conferences to attend, existing projects to maintain, and other tasks that make up a standard work week.

Accurate estimates are difficult because of the imprecise nature of most software projects. In the initial phase of a project, complete requirements are not known, and the way you will implement those requirements is even less clear. As you clarify the objectives and implementation plans, you can make more realistic estimates.

Some of the current best practice estimation techniques in software engineering are described in the following sections. All of them require breaking the project down into more manageable components that can then be estimated individually. There are other methods of estimating development time. See Appendix A, *References*, for a list of documents that describe these and other estimation techniques in more detail.

Lines of Code/Number of Nodes Estimation

Software engineering documentation frequently refers to *Lines of Code* (LOC) as a measurement, or metric, of software complexity. Lines of Code as a measurement of complexity is very popular in part because the information is easy to gather. Numerous programs exist for analyzing textual languages to measure complexity. In general, Lines of Code measurements include every line of source code developed for a project, excluding comments and blank lines.

The VI Metrics tool, described in Chapter 8, *VI Metrics Tool*, provides a method for measuring a corresponding metric for G-based code. The VI Metrics tool lets you count the number of nodes used within a VI or

within a hierarchy of VIs. A node is almost any object on a block diagram excluding labels and graphics, but including functions, VIs, and structures such as loops and sequences. See the Chapter 8, *VI Metrics Tool*, for details on how to use this tool and more information on the accounting mechanism it uses.

You can use *Number of Nodes* as a method for estimating future project development efforts. For this to work, you must build up a base of knowledge about current and previous projects. You must have an idea of the amount of time it took to develop components of existing software products and associate that information with the number of nodes used in that component.

Armed with that historical information, you next need to estimate the number of nodes required for a new project. It is not possible to do this for an entire project at once. Instead, you must break the project down into subprojects that you can compare to other tasks completed in the past. Once you have broken it down, you can estimate each component and produce a total estimate of both the number of nodes and the time required for development.

Problems with Lines of Code and Numbers of Nodes

Size-based metrics are not uniformly accepted in software engineering. Many people favor them because it is a relatively easy metric to gather and because a lot of literature has been written about it. Detractors of size metrics point out the following flaws:

- Size-based metrics are organization-dependent. Lines of code/numbers of nodes can be useful within an organization as long as you are dealing with the same group of people and they are following the same style guidelines. Trying to use size metrics from other companies/groups can be very difficult because of differing levels of experience, different expectations for testing and development methodologies, and so on.
- Size-based metrics are also programming language–dependent. Comparing a line of code in assembly language to one written in C can be like comparing apples to oranges. Statements in higher-level languages can provide more functionality than those in lower-level languages. Comparing numbers of nodes in G to lines of code in a textual language can be inexact for this reason.
- Not all lines of code are created with the same level of quality. A VI that retrieves information from a user and writes it to a file can

be written so efficiently that it involves a small number of nodes or it can be written poorly with a large number of nodes.

- Not all lines of code are equal in complexity. An add function is much easier to use than an array index node. A block diagram that consists of 50 nested loops is much more difficult to understand than 50 icons connected together in a line.
- Size-based metrics rely on a solid base of information associating productivity with various projects. To be accurate, you should have statistics for each member of a team because the experience level of team members varies.

Despite these problems, size metrics are used widely for estimating projects. A good technique to use is to estimate a project using size metrics in conjunction with one of the other methods described later in this chapter. The two different methods can serve as checks for each other. If you find differences between the two estimates, analyze the assumptions in each to determine the source of the discrepancy.

Effort Estimation

Effort estimation is similar in many ways to number of nodes estimation. You break the project down into components that can be more easily estimated. A good rule of thumb is to break the project into tasks that take no more than a week to complete. Tasks that are more complicated are difficult to estimate accurately.

Once you have broken the project down into tasks, you can estimate the time to complete each task and add the results to calculate an overall cost.

Wideband Delphi Estimation

You can use wideband delphi estimation in conjunction with any of the other estimation techniques to achieve more reliable estimates. For successful wideband delphi estimation, multiple developers must contribute to the estimation process.

First divide the project into separate tasks. Then meet with other developers to explain the list of tasks. Avoid discussing time estimates during this early discussion.

Once you have agreed on a set of tasks, each developer separately estimates the time it will take to complete each task, using, for example, uninterrupted person-days as the unit of estimation. The developers

should list any assumptions made in forming their estimates. The group then reconvenes to graph the overall estimates as a range of values. It is a good idea to have a person outside the development team lead this meeting and keep the estimates anonymous.

After graphing the original set of values, each developer reports any assumptions made in determining the estimate. For example, one developer might have assumed a certain VI project takes advantage of existing libraries. Another might point out that a specific VI is more complicated than expected because it involves communicating with another application or a shared library. Another team member might be aware of a task that involves an extensive amount of documentation and testing.

After stating assumptions, each developer reevaluates and adjusts the estimates. The group then graphs and discusses the new estimates. This process might go on for three or four rounds.

In most cases, you will converge to a small range of values. Absolute convergence is not required. After the meeting, the developer in charge of the project can use the average of the results, or he or she might ignore certain outlying values. If some tasks turn out to be too expensive for the time allowed, he or she might consider adding resources or scaling back the project.

Even if the estimate is incorrect, the discussion from the meetings gives a clear idea of the scope of a project. The discussion serves as an exploration tool during the specification and design part of the project so you can avoid problems later.

For a list of documents that include more information on the wideband delphi estimation method, see Appendix A, *References*.

Other Estimation Techniques

Several other techniques exist for estimating development cost. These are described in detail in some of the documents listed in Appendix A, *References*. The following list briefly describes some other popular techniques.

- Function Point Estimation—Function-point estimation differs considerably from the size estimation techniques described so far. Rather than divide the project into tasks that are estimated separately, function points are based on a formula applied to a category breakdown of the project requirements. The requirements

are analyzed for features such as inputs, outputs, user inquiries, files, and external interfaces. These features are tallied, and each is weighted. The results are added up to produce a number representing the complexity of the project. Then this number can be compared to function-point estimates of previous projects to determine an estimate.

Function-point estimates were designed primarily with database applications in mind but have been applied to other software areas as well. Function-point estimation is popular as a rough estimation method because it can be used early in the development process based on requirements documents. However, the accuracy of function points as an estimation method has not been thoroughly analyzed.

- COCOMO Estimation—COCOMO (COnstructive COst MOdel) is a formula-based estimation method for converting software size estimates to estimated development time. COCOMO is a set of methods that range from basic to advanced. Basic COCOMO makes a rough estimate based upon a size estimate and a simple classification of the project type and experience level of a team. Advanced COCOMO takes into account reliability requirements, hardware features and constraints, programming experience in a variety of areas, and tools and methods used for developing and managing the project.

Mapping Estimates to Schedules

An estimate of the amount of effort required for a project can differ greatly from the calendar time needed to complete the project. You might accurately estimate that a VI should take only two weeks to develop. However, in implementation you must fit that development into your schedule. You might have other projects to complete first, or you might need to wait for another developer to complete his or her work before you can start the project. You might have meetings and other events during that time.

Estimate project development time separately from scheduling it into your work calendar. Consider estimating tasks in *ideal* person-days that correspond to eight hours of development without interruption.

After estimating project time, try to develop a schedule that accounts for overhead estimates and project dependencies. Remember that you

have weekly meetings to attend, existing projects to support, reports to write, and so on.

Record your progress at meeting both time estimates and schedule estimates. Track project time and time spent on other tasks each week. This information might vary from week to week, but you should be able to come up with an average that is a useful reference for future scheduling. Recording more information helps you plan future projects accurately.

Tracking Schedules Using Milestones

Milestones are a crucial technique for gauging progress on a project. If completing the project by a specific date is important, consider setting milestones for completion.

Set up a small number of major milestones for your project, making sure that each one has very clear requirements. Because the question "Did you reach the milestone?" only can be answered "yes" or "no". An answer of "mostly" or "90% of the project is complete" is meaningless. In the case of the 90% answer, the first 90% might have been completed in two months while the remaining 10% will require another year.

To minimize risk, set milestones to complete the most important components first. If, after reaching a milestone, the schedule has slipped and there is not enough time for another milestone, the most important components will have been completed.

Throughout development, strive to keep the quality level high. If you defer problems until a milestone is reached, you are in effect deferring risks that might delay the schedule. Delaying problems can make it seem like you are making more progress than you actually are. Also, it can create a situation where you attempt to pile new development on top of a house of cards. Eventually the facade comes crumbling down, and you waste more time and resources than you would have if you had dealt with the problem in the first place.

When working toward a major milestone, set smaller goals to gauge progress. Derive minor milestones from the task list created as part of your estimation.

Major and minor milestones are discussed in depth in a number of the books listed in Appendix A, *References*.

Responding to Missed Milestones

One of the biggest mistakes people make is to miss a milestone but not reevaluate the project as a consequence. After missing a milestone, many developers continue on the same schedule, assuming that they will work harder and be able to make up the time.

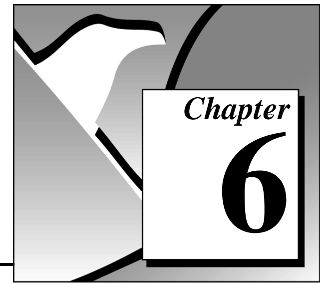
Instead, if you miss a milestone you should evaluate the reasons you missed it. Is there a systematic problem that could affect subsequent milestones? Is the specification is still changing? Are quality problems slowing down new development? Is the development team at risk of burning out from too much overtime?

Consider problems carefully. Discuss each problem or setback and have the entire team make suggestion on how to get back on track. Avoid accusations. You might have to stop development and return to design for a period of time. You might decide to cut back on certain features, stop adding new features until bugs are fixed, or renegotiate the schedule.

Address problems as they arise. Then monitor progress to avoid repeating mistakes or making new ones.

Missing a milestone should not come as a complete surprise. Schedule slips do not occur all at once; they happen little by little, day by day. Correct problems as they arise. Do not wait until the end of the milestone or the end of the project. If you do not realize that you are behind schedule until the last two months of a year-long project, there probably will not be anything you can do to get back on schedule.

Creating Documentation



This chapter describes techniques for documenting your software.

You need to create several documents for software that you develop. There are two main categories for this documentation:

- Design-related documentation—Requirements, specifications, detailed design plans, test plans, and change history documents are all examples of the kinds of design-related documents that you might need to produce.
- User documentation—User documentation consists of printed manuals and online help files that explain how to use your software.

The style of each of these documents can be very different. Design documentation generally is written for an audience with quite a bit of prior knowledge of the tools they are using. User documentation is written for an audience with a lesser degree of understanding.

The size and style of each document can vary according to the type of project you are working on. For simple tools that only will be used in-house, you might not need to do very much of either. If you plan to sell a product, you must allow a significant amount of time to develop detailed user-oriented documentation that describes the product. For products that must go through a quality certification process, such as an FDA review, you must ensure that the design-related documentation is as detailed as required.

Developing Design-Related Documentation

The format and detail level of the documentation you develop for requirements, specifications, and other design-related documentation is determined by the quality goals of your project. If you are developing to meet a quality standard such as ISO 9000, the format and detail level of these documents are very different than an in-house project.

For a complete description of the types of documents you should prepare as part of your development process, refer to one of the resources in Appendix A, *References*.

There are features in LabVIEW and BridgeVIEW that can help you produce some of the documentation you must create. Some of the features in this toolkit that you can use to simplify the process are listed below.

- History window—The History window is a place to record changes to a VI as you make them. When you check in a file using the Source Code Control tools described in Chapter 11, *Source Code Control Tools*, the History window text is retained by the SCC tools. You can view it later or print it using the report generation features of the SCC tools.
- SCC report generation—In addition to accessing the change history for a file, you can view the change history for all files under Source Code Control to see which files have changed and when. You also can view listings of the projects under SCC and the files that make up those projects. This information either can be viewed on screen or saved to a file so that you can import it into a word processor to add it to reports. See the description of the SCC Advanced dialog box in Chapter 11, *Source Code Control Tools*, for more details.
- Print documentation dialog box—With this dialog box, you can create printouts of the front panel, block diagram, connector pane, and description of a VI. It also prints out the names and descriptions of controls and indicators for the VI and the names and paths of any subVIs. You can print this information or save it to a file.
- Print hierarchy tool—With this tool, you can automate the process of printing documentation for the VIs in your VI hierarchy. The print hierarchy tool is described in Chapter 9, *Print Hierarchy Tool*.

Developing User Documentation

The format of user documentation depends on the type of product you create.

Documentation for a Library of SubVIs

If the software you are creating is a library of subVIs for use by other developers, such as an instrument driver or add-on package, you should create documents with a format similar to the *LabVIEW Data Acquisition VI Reference Manual* (LabVIEW Users) or Appendix A, *MMI Function*

Reference, in the *BridgeVIEW User Manual* (BridgeVIEW Users). Because the audience is other developers, you can assume they have a working knowledge of LabVIEW or BridgeVIEW. Your documentation might consist of an overview of the contents of the package, examples of how to use the subVIs, and a detailed description of each subVI.

For each subVI, you might want to include the VI name and description, a picture of the connector pane, and the description and a picture of the data type for each of the controls and indicators on the connector pane.

You can generate much of this documentation easily if you use the description feature for VIs and controls as described in the *VI and Control Descriptions* section of this chapter. You can use **Print Documentation** to create a printout of a VI in a format that is almost identical to the format used in the VI Reference manuals that ship with BridgeVIEW and LabVIEW.

If you want to incorporate the text into a manual or help file, you can use **Print Documentation** and then select **Save as Text** to save the text information to a file. The text file produced includes the VI name, description, all control and indicator data types and descriptions, a list of subVIs used by this VI, and the VI history. You can use a word processing program to format this text and use a screen capture program to capture the connector pane picture from the help window and the block diagram terminals for the data types. Screen capture programs can be found at many FTP locations.

A simple way to manage the data type pictures is to create a separate document containing a complete set of terminal icons. You can keep this document open at the same time you document your VIs. When you need a terminal description, you can copy and paste the icons.

Documentation for an Application

If you are developing an application for users who are not familiar with LabVIEW or BridgeVIEW, your documentation will have a very different format. Your documentation should cover basic features such as installation and system requirements. It should provide an overview of how the package works. If the package uses I/O, describe the necessary hardware and any configuration that must be done before starting your application.

For each front panel the user interacts with, provide a picture of the front panel and a description of the major controls and indicators.

Organize the front panel descriptions in a top-down fashion, with the first front panels that the user sees documented first. As with the previous section, you can use the Print Documentation dialog box to access the VI and control names and descriptions. These can be used in conjunction with a screen capture program to document your VIs effectively.

Creating Help Files

You can create your own online help or reference documents if you have the right development tools. Online help documents are based on formatted text documents. You can create these documents using a word processing program, such as Microsoft Word, or using one of the tools described later in this section. Special help features such as links and hotspots are created as hidden text.

You can use the same Print Documentation dialog box **Save as Text** feature described earlier to help in creating the source material for your help documents. Once you have the text for a VI in a text file, you can add links and graphics to make it interactive.

Once you have created source documents, use a help compiler to create a help document. If you need help files on multiple platforms, you must use the help compiler for the specific platform on which they will be used. You might want to use any of the following compilers. The Windows compilers also include tools for creating help documents.

- **(Windows)** RoboHelp from Blue Sky Software, (800) 677-4946; for international customers (619) 459-6365
- **(Windows)** Doc-To-Help from WexTech Systems, Inc., (800) 939-8324.
- **(Macintosh)** QuickHelp from Altura Software, (408) 655-8005.
- **(UNIX)** HyperHelp from Bristol Technologies, (203) 438-6969

Once you have created and compiled your help files, you can add them to the **Help** menu of LabVIEW, BridgeVIEW, or your own custom application by placing them in the Help directory. You also can link to them directly from a VI in one of two ways:

- You can add a link using the **VI Setup»Documentation** option. Pop up on the VI connector pane of the VI for which you want to link a file and select **VI Setup»Documentation**. Select the Help Tag box and type the topic you would like to link to in the help file. Choose

the help file by clicking the **Browse...** button. The path of the file appears in the Help Path box. Then this link can be accessed from the Help Window. Also, if the VI is a subVI on another block diagram, you can pop up on the subVI icon and select **Online Help** to jump to the selected topic in the specified help file.

- You can use the Help functions from the **Functions»Advanced»Help** palette to jump to topics in specific help files programmatically.

VI and Control Descriptions

VI Description

The VI description in the **Get Info** dialog box from the File menu is often a user's only source of information about a VI. A VI description is displayed in the Help window when the user places the mouse cursor on its icon, either the icon on the VI front panel or the icon used as a subVI in a block diagram.

Important items to include in a VI description are as follows:

- An overview of the VI function, followed by as much detail as you can supply.
- Instructions for use.
- Description of inputs and outputs.

Self-Documenting Front Panels

One way of providing important instructions is to place a block of text prominently on the front panel. A concise enumerated list of important steps is invaluable. You might even put a suggestion there that says, "Select **Get Info** from the File menu for instructions" or "Select **Show Help** from the Help Menu." For long instructions, you can use a scrolling string instead of a free label, but be sure to pop up and select **Make Current Value Default** to save the text.

If that requires too much space on your front panel, you can put a highly visible HELP button on the front panel instead. Put the instruction string on its own front panel that pops up when the user pushes the help button. Use the window setup options in **VI Setup** to configure this help panel as either a dialog box, requiring the user to press an **OK** button to

close it and continue, or as a window that can be moved anywhere and closed anytime.

Alternatively, you can use this help button to open an entry in an online help file. You can use the Help functions from the **Functions» Advanced»Help** palette to open the LabVIEW or BridgeVIEW Help window or to open a help file and jump to a specific topic.

Control and Indicator Descriptions

Include a description for every control and indicator. You can enter this with the **Description** popup menu item. An object description is shown in the Help window when the user places the mouse cursor over the object.

When confronted with a new VI, a user has no alternative but to guess the function of each control and indicator unless you include a description. Always remember to enter a description as soon as you create the object. Then, if you copy the object to another VI, the description is copied also. Also be sure to tell users about this feature.

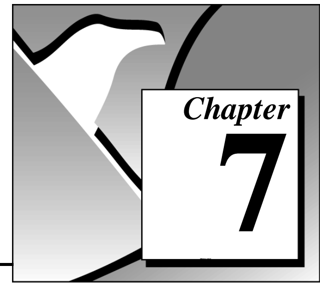
Every control and indicator needs a description that includes the following information:

- Functionality
- Data type
- Valid range (for inputs)
- Default value (for inputs)
- Behavior for special values (0, empty array, empty string...)
- Additional information, such as whether the user must set this value always, often, or rarely

Alternatively, the default value can be listed in parentheses as part of the VI name. For controls and indicators that are on the VI connector pane, mark the inputs and outputs as required, recommended, or optional. See the *Connector Panes* section in Chapter 7, *Using Consistent Style—The G Style Guide*, for more information.

 © National Instruments Corporation

Using Consistent Style—The G Style Guide



This chapter describes some recommended practices for good programming technique and style. Remember that these are only *recommendations,* not laws or strict rules. Consider your audience—users need a good front panel; developers need a good block diagram; and everybody needs good documentation. Several experienced G programmers have contributed to this guide.

As mentioned in Chapter 3, *Incorporating Quality into the Development Process*, inconsistent style causes problems when multiple developers are working on the same project. The resulting VIs can confuse users and be difficult to maintain. To avoid these problems, establish a set of style guidelines for VI development. You can establish an initial set at the beginning of the project and add additional guidelines as the project progresses.

A style checklist is included at the end of this chapter to help you maintain consistency and quality as you develop VIs. To save time, review the list before and during development.

Hierarchy on Disk

The layout of your VIs on disk should reinforce the hierarchical organization of your software. Make the top-level VI(s) directly accessible. Place subVIs in subdirectories and group them to reflect any modular components you have designed, such as instrument drivers, configuration utilities, and file I/O drivers.

Create a directory for all of the VIs for one application (pictured in the following figure as a Macintosh folder), and give it a meaningful name. Save the main VI(s) in this directory and the subVIs in a subdirectory.

If the subVIs have subVIs, continue the directory hierarchy downward like an inverted tree.

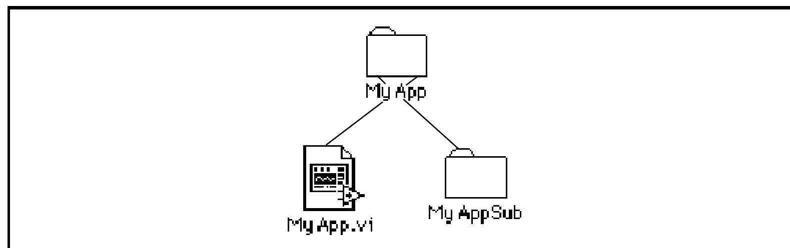


Figure 7-1. A Directory Hierarchy

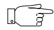
When naming VIs, VI libraries, and directories, avoid using characters that are not accepted by all file systems, such as slash (/), backslash (\), colon (:), tilde (~), and so on. With the exception of Windows 3.1, most operating systems accept long descriptive names for files (up to 31 characters on a Macintosh, 255 characters on other platforms). See the *Multiplatform Issues* section of Chapter 11, *Source Code Control Tools*, for more details on file name limits for different platforms.

Check preferences (**Edit»Preferences**) to make sure that the VI Search Path contains <topvi>* and <foundvi>* (the * causes all subdirectories to be searched). In the previous example `MyApp.vi` is the top VI. This means that the application will search for subVIs in the directory `MyApp` before searching the entire disk, and that once a subVI is found in a directory, the application will look in that directory for subsequent subVIs.

Avoid creating files with the same name anywhere within your hierarchy. Only one VI of a given name can be in memory at a time. If you have a VI with a specific name in memory, and you attempt to load another VI that references a subVI of the same name, the VI will link to the VI in memory. If you make backup copies of your files, be sure to save them into a directory outside the normal search hierarchy.

Hierarchy with VI Libraries

If you need to create an application or ship VIs to a customer using Windows 3.1, save the VIs into VI libraries (LLBs). Within LLBs, the VIs can have long, descriptive names even under Windows 3.1 (only the LLB itself and the directories will be subject to the 8+3 character limit).



Note: ***LLBs are not supported by the G Source Code Control tools described in*** *Chapter 11, Source Code Control Tools****. As described in that chapter, in the section*** *Using Individual Files Instead of VI Libraries (LLBs)****, you can develop under Windows 95 or Windows NT using directories. When it comes time to test and ship under Windows 3.1, you can save your files into LLBs.***

There are some disadvantages to saving VIs in a VI library. First, as a VI library grows, it takes longer to save VIs to it because a copy of the entire library must be made during the save.

Second, VIs inside a VI library are not visible to your computer file management system so they cannot be found using the operating system's **Find File** command.

The third disadvantage to LLBs is the lack of hierarchy within a VI library. You can simulate one level of hierarchy by marking some VIs as top-level VIs using the **File»Edit VI Library** dialog box. Top-level VIs are listed above and apart from the others in the **Open File** dialog box as shown in the following figure.

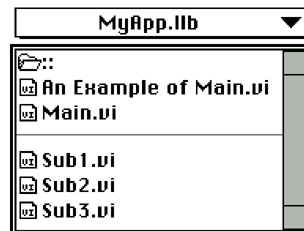


Figure 7-2. Top-Level VIs Listed at the Top of a VI Library

If you use LLBs, use a combination of directories and VI libraries to get the advantages and avoid the disadvantages of both. Separating main

VIs and subVIs into two or more VI libraries in the same directory makes the VI libraries smaller and the hierarchy more obvious.

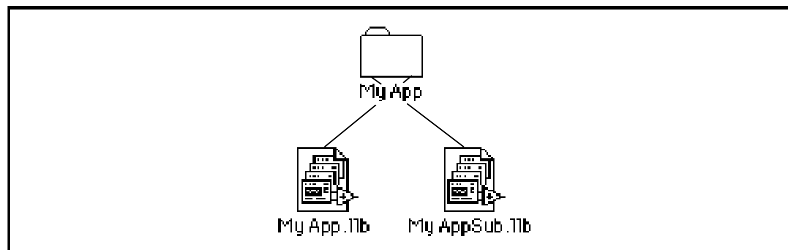


Figure 7-3. A Mixture of Directories and VI Libraries

You might move the LLBs containing subVIs into subdirectories to distinguish the top-level VIs from the subVIs. You can break the subVIs into multiple LLBs without making the top-level structure too confusing.

Front Panels with Style

Consider the following analogy:

> The front panel of a VI is to a G program what the cockpit is to an airplane. Just as cockpit instruments give the pilot control over even the most technologically complex aircraft, G front-panel instruments give you, the programmer, control over program input and output. No conventional programming environment has anything comparable to LabVIEW's and BridgeVIEW's built-in user interface.

How do you make the best use of this powerful interface? A user's first contact with your work (and with LabVIEW or BridgeVIEW) is the *front panel*, so it had better be *good*.

Consistency

Even if you decide not to follow these guidelines, at least be consistent. The user cannot adapt to your style if it contains disconcerting changes with every front panel. While stylish fonts and garish colors are eye-catching, they distract the user. Standardize on a few colors, fonts, and layout practices that are attractive and functional. Professional societies have written standards for man machine interface design.

Text

Do not be tempted to use all of the fonts and styles available. Stick to three standard fonts—application, system, and dialog—unless you have a specific reason to use a different font. For example monospace fonts (fonts that are not proportionally-spaced) are useful for string controls and indicators where the number of characters is critical. To set the default font, choose it from the **Text** menu *without any text or objects selected*. You can select all labels that need changing and set the font in all at once using the **Text** menu.

The actual font used for the three standard fonts varies depending on the platform, your preferences, and video driver settings (when working under Windows). Text might appear larger or smaller. To compensate for this, allow extra space for larger fonts and keep the **Size to Text** option on. Use carriage returns to make multi-line text instead of resizing the text frame. When designing for multiple computers, prevent front panels from becoming too busy and allow extra space between controls. By allowing space, you can keep controls and indicators from overlapping if they grow because of font changes. It is a best to keep front panels simple anyway, to avoid confusing the user. Fonts are the least portable aspect of the front panel, so always test them on all of your target platforms.

Some suggestions for a consistent set of text styles are listed below:

Indicator and Control labels

- **Application Font Bold** or Dialog Font for controls and indicators of primary importance; generally, these are on the connector.
- Application Font Plain for things like secondary indicators, or controls used as constants.

Groups of Controls or Titles

- Dialog Font looks more important than application font, without being too distracting.

Indicators and controls on pop-up panels

- Dialog Font makes homemade dialog boxes look more natural for your platform.

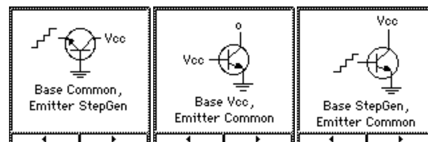
Color

Like fonts, it is easy to get carried away with color. The particular danger of color is that it distracts the operator from important information. For instance, a yellow, green, and shocking orange background make it difficult to see a red *danger* light. Another problem is that other platforms might not have as many colors available. Also, some users have black and white monitors, which cannot display certain color combinations very well. For example, on a black-and-white monitor, black letters on a red background is displayed as all black. Use a minimal number of colors, with lots of black, white and gray. The following are some simple guidelines for using color:

- Never use color as the sole indicator of device state. People with some degree of color-blindness (5% of men) might not detect the change. Also, multiplot graphs and charts can lose meaning when displayed in black and white—use line styles in addition to color.
- Backgrounds should be light gray, white or light pastel colors.
- Select bright, highlighting colors only when the item is very important, such as an error notification.
- Always check your VI on other platforms and on a black-and-white monitor.
- Most of all, be consistent.

Graphics and Custom Controls

Enhance the functionality of your front panel with imported graphics. You can import bitmaps, Macintosh PICTs, Windows Enhanced Metafiles, and text objects for use as backgrounds or in Pict Rings and custom controls.



Use a Pict Ring when a function or mode is conveniently described by a picture.

A custom Boolean control that is transparent in one state appears when the state changes. A completely transparent Boolean is useful for detecting mouse clicks in specified regions of the screen.

Check how your imported pictures look when your VI is loaded on another platform. For example, a Macintosh PICT file that has an irregular shape might convert to a rectangular bitmap with a white background on the PC or UNIX.

One disadvantage of imported graphics: They slow down screen updates. Here are some tips to improve performance:

- Make sure indicators and controls are not placed on top of a graphic object. That way, the object does not have to be redrawn each time the indicator is updated.
- If you must use a large background picture with controls on top of it, try breaking it into several smaller objects and import them separately. Large graphics usually take longer to draw than small ones. For instance, you could import several pictures of valves and pipes individually instead of importing one large picture.

Front Panel Layout

Consider the arrangement of controls on front panels. For top-level VIs that users see, place the most important controls in the most prominent positions. Use **Align and Distribute** to make a nice, regular layout. Use **Set Panel Order** from the **Edit** menu to arrange controls in a logical sequence (see the *Key Navigation* section of this chapter for more information). Do not overlap controls with other controls or with their own label, digital display, or other part, unless you are trying to achieve some special effect. Overlapped controls are much slower to draw, and flash (erase first). Place any **Start** or **Stop** buttons near the **Run** button on the tool bar. Do this for two reasons: The buttons are easier to find and click, and the **Stop** button will be more prominent than the **Abort** button (if you did not hide it), and the user is less likely to abort the VI by accident.

Use simple elements such as rounded rectangles to visually group objects with related functions. Use clusters to group related data. However, do not use clusters for aesthetic purposes only; it makes connections to your VI more difficult to understand. Avoid importing graphic objects that are inanimate copies of real controls. For instance, do not use a copy of a cluster border to group controls that are not actually in a cluster.

For subVI front panels that the user does not see, the objects can be placed so they correspond to the connector pattern. Generally, inputs should be on the left and outputs on the right.

Sizing and Positioning Front Panels

Front panels should fit on a monitor that is the standard size for most intended users. Shrink the window as much as possible without crowding controls or sacrificing a good layout. If your VIs are intended for in-house use and everyone has a large monitor, go ahead and design large front panels. If you are doing commercial development, keep in mind that not everyone has a large monitor.

Front panels should open in the upper-left hand corner of the screen for the convenience of users with small screens. Sets of VIs that are often opened together should be placed so the user can see at least a little bit of each. Place front panels that open automatically in the center of the screen. Check the **Auto-Center** option in **VI Setup** to optimize this for monitors of various sizes.

Moving a window is not considered a modification within the VI editor. To save the VI with the windows properly placed, make a small change (like moving a control by one pixel, then back) and save the VI, or select **Save As...** and use the same name.

Controls and Indicators

Descriptions

Every control and indicator should have a description. See the *VI and Control Descriptions* section in Chapter 6, *Creating Documentation*, for more details.

Labels

Labels are displayed in the Help window as part of the connector. Label the most important controls and indicators on a front panel in **boldface**. Display controls and indicators that are rarely used in square brackets. If the default value of a control is valid, add it to the name in parentheses. Include the units of the value, where applicable. The appearance of the inputs and outputs in the help window is affected by the Required/Recommended/Optional setting, which is described in the *Connector Panes* section of this chapter.

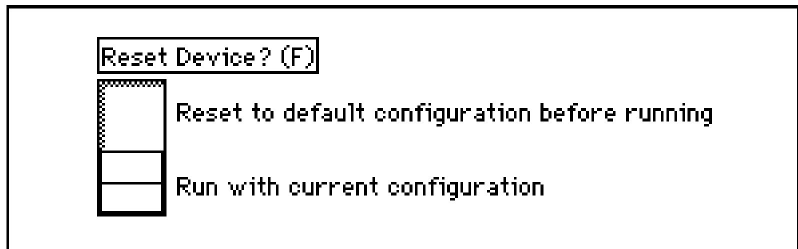
The name of a control or indicator should describe its function. For example, for a ring or labeled slide with options for Volts, Ohms, or Amperes, a name like “Select units for display” is better than “V/O/A”

and is certainly an improvement over the generic "Mode." Of course, long names use valuable space on the block diagram, especially if you use any local variables or **Bundle/Unbundle by Name** functions. You might prefer to give the control a short name, then add an explanatory label to it.

For Booleans, the name should give an indication of which state corresponds to which function, while still indicating the default state. A recommended format for a Boolean where true means reset, but the default is false is:

Reset Device? (F)

Free labels next to the Boolean can help clarify the meaning of each position on a switch, as shown in the following figure.



Enumerations vs. Rings

Rings and enumerations look identical on a front panel, but they are different. On a block diagram, a ring is simply an integer numeric. Rings have the appearance of a pop up menu, associating each string with a number. The strings can be set at edit time or at run time using an attribute node.

An enumeration is similar to a ring, but the strings in the enumeration are really a part of the enumeration's data type. If you wire an enumeration to a case structure, the case structure displays the names from the enumeration instead of the numbers. Also, if you pop-up on an enumeration input of a function or subVI and create a control, constant, or indicator, the resulting object will also be an enumeration (with a ring, you would simply get a numeric).

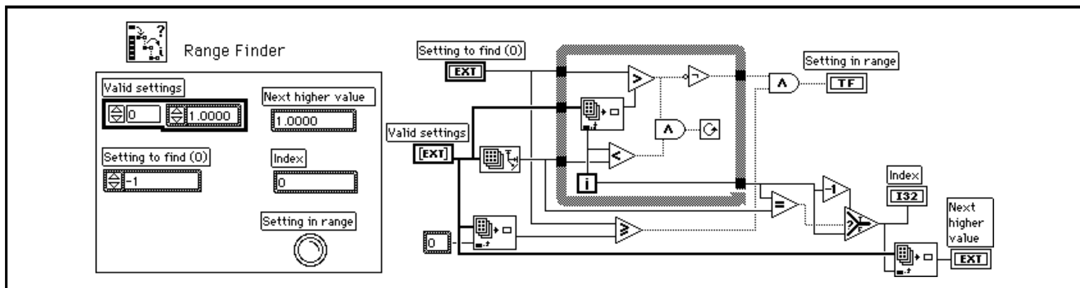
Because the names are really a part of the type, you cannot change the names in an enumeration programmatically at run-time. Also, you cannot compare two enumerations of different types. If you wire an

enumeration to something that expects a standard numeric, you will see a coercion dot because the type is being converted.

Enumerations are useful for making code easier to read. Rings are useful for front panels that the user interacts with, where you want to programmatically change the strings.

Default Values, Ranges, and Coercion

Expect the user to supply invalid values to every control. You can check for invalid values in your block diagram, or set the control **Data Range** item to coerce values into the desired range (minimum, maximum, and increment). If the values are not evenly spaced (such as a 1-2-5 sequence) use a function similar to the Range Finder VI shown below.



Other difficult situations must be handled programmatically. Many GPIB instruments limit the permissible settings of one control based on the settings of another. For example, a voltmeter might permit a range setting of 2000 V for DC, but only 1000 V for AC. If the affected controls like Range and Mode reside in the same VI, put the interlock logic there (see the *Attribute Nodes* and *Local Variables* sections later in this chapter). If one or more of the controls are not readily available, you can request the present settings from the instrument to ensure that you do not try to set an invalid combination.

There are some speed and memory usage drawbacks to limiting ranges. The **Data Range** function adds some execution overhead, as does the Find Range VI and similar VIs. If you choose **Suspend** for range error action, the VI front panel remains in memory and will open if a range error occurs. This consumes additional memory.

Controls should have reasonable default values. A VI should not fail when run “as-opened” with default values. Remember to show the default in parentheses in the control’s label. Do not set default values of

indicators like graphs, arrays, and strings without a good reason; that wastes disk space when saving the VI.

Make intelligent use of default values. In the case of the high-level file VIs such as the Write Characters to File VI, the default is an empty path that forces a file dialog. This can save the use of a Boolean switch in many cases.

Attribute Nodes

Use attribute nodes to give the user more feedback on the front panel. There are many things you can do to make your VI easier to use. Some ideas include:

- Set the text focus to the main, most commonly used control.
- Gray out or hide controls that are not currently relevant or valid.
- Guide the user through steps by highlighting controls.
- Change screen colors to bring attention to error conditions.

Key Navigation

Some users prefer to use the keyboard instead of a mouse. In some environments, such as a manufacturing plant, only a keyboard is available. Even if a mouse is used, keyboard shortcuts, such as using the <Enter> key to select the default action of a dialog box, add convenience.

For these reasons, consider including keyboard shortcuts to your programs.

Consider the tab order of controls. If you select **Edit»Panel Order**, you can see the order of your front panel controls. This order controls the tab order for your front panel. Set the order to read left-to-right and top-to-bottom.

Pay attention to the key navigation options for buttons on the front panel. Key navigation options can be set from the **Key Navigation** pop up of any control. Set the <Enter> key to be the keyboard shortcut to a front panel's default control. The only exception to this rule is that if you have a multiline string control on the front panel, you might not want to use the <Enter> key as a shortcut.

If your front panel has a **Cancel** button, assign a shortcut to the <Escape> key. You also can use function keys as navigation buttons to move from screen to screen. If you do this, be sure to use the shortcuts consistently. Do not use F5 on one front panel and F6 on another front panel for the same action.

For controls that are offscreen, use the key navigation dialog box to skip over the controls when tabbing.

Also, you might consider using the Key Focus attribute to set the focus programmatically to a specific control when the front panel first is opened.

Local Variables

If you have controls with interdependent values, use local variables to keep the values consistent and valid. For example, a VI that generates a square wave might have two inputs—period and frequency. If the user sets period, the VI should detect the change in value and change frequency to the corresponding value.

Use local variables when you need a control/indicator combination. For example, the VI might set some parameter values (write to controls with local variables), but the user must be able to override those values by entering his or her own (type into controls).

But, avoid using local variables if possible. Some users use local variables because it seems like a convenient way to avoid passing wires from one point to another on your block diagram. The problem is that it hides the data flow, making it more difficult to understand and maintain the block diagram code. Also, it makes it easy to have *race conditions*, in which multiple locations on the block diagram attempt to modify the same local, resulting in the loss of data. See Chapter 26, *Performance Issues*, of the *G Programming Reference Manual*, or Chapter 27, *Performance Issues*, of the *LabVIEW User Manual* for more details about using local variables.

VI Setup

Consider the items in the VI Setup dialog box carefully (pop up on the VI icon to access VI Setup).

Think about the window behavior and style of every VI in your project. Check **Show Front Panel when Called** and **Close Afterwards if Originally Closed** for front panels that you want to appear and disappear automatically. Check **Dialog Box** for front panels that should wait for input from the user before the program can continue. Turn off the **Allow User to Close Window** option to keep users from accidentally closing an important front panel while it is running. Disable the scroll bars, the menu bar, and the tool bar unless the user needs them. Remember that you still can abort a VI by using the following keyboard shortcuts:

- <Ctrl-period> **(Windows)**
- <Cmd-period> **(Macintosh)**
- <meta-period> **(Sun)**
- <Alt-period> **(HP-UX)**

You still can use the keyboard shortcuts for cut, copy, and paste even if the menu is hidden.

Hide the **Abort** button if the user should not abort the VI—it is always best to provide a front-panel Boolean **STOP** button for VIs that loop. Hiding the **Abort** button disables the keyboard shortcut for aborting the VI. You can hide the single-stepping and execution highlighting buttons to save a small amount of execution time when the VI is finished; but these debugging tools are often useful to a user who is trying to understand how the block diagram works.

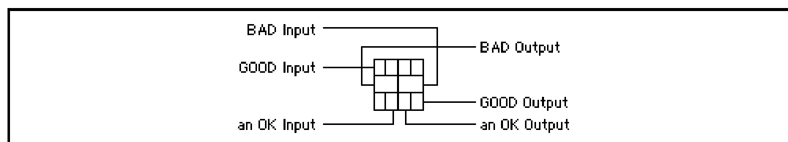
Do not set higher priority than the default on any VI without giving it some serious thought. A high-priority VI that loops forever will block execution of all other VIs. See Chapter 24, *Understanding the G Execution System*, of the *G Programming Reference Manual*, or Chapter 26, *How LabVIEW Executes VIs*, of the *LabVIEW User Manual* for information on how priorities work.

Connector Panes

Consider selecting a connector pattern with extra terminals. You can leave these extra terminals unconnected. That way, you do not have to change the connector pattern for your VI if you find that you need another input or output later on. Changing patterns requires replacement of the subVI in all calling VIs. By adding extra, unused terminals, you can add an input or output with minimal effect on your hierarchy.

Put at least one input and one output on each subVI, for the sake of defining data flow. Error in and error out are ideal data flow connections. If a set of VIs is used together and must be sequenced, you can add a common thread. See the *Adding Common Threads* section later in this chapter for more information.

Make connections for inputs on the left and outputs on the right. This conventional left-to-right data flow prevents complicated, unclear wiring patterns.



When the same inputs and outputs are used in several VIs, try to place them in the same location on each VI. For example, refnums are usually located at the top left and right of an icon, and error I/O are located at the bottom left and right. Doing this makes it easier to wire icons together.

On the front panel, you can edit required inputs for subVIs by clicking on a terminal in the connector pane at the upper right side of the window and choosing **This Connection is»**. If the connector pane is not visible, select **Show Connector Pane** first. From the **This Connection is»** submenu, select **Required**, **Recommended**, or **Optional**. By default, inputs are all considered to be Recommended.

If you have the **Show Warnings** preference enabled, the error window will warn you of unwired, recommended inputs.

If you designate an input as Required, it must be wired in a calling VI for the VI to work. Also, required inputs will appear in bold in the Help window. This is appropriate for inputs such as refnums where the VI does not make sense if the input is not wired. You should not make an input Required unless it is necessary for the VI to execute properly.

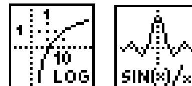
If you make an input Optional, the Help window does not display it in simple help mode, which helps to keep the connector pane in the help window from becoming too cluttered. With simple help mode turned off, the input appears grayed out. You should use the Optional setting for parameters that you rarely need to wire.

You can specify whether outputs should be recommended or optional, but you cannot mark outputs as required.

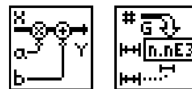
Icons

Create a meaningful icon for every VI. Always create a black and white icon (for printouts and menus), and add color later. The examples and vi.lib directories are full of well-designed icons that illustrate the functionality of the underlying program. Collect ideas for icons. You might have to use text if you cannot create a picture; however, if you intend to send your VIs to customers who speak other languages, a well-chosen icon is much more effective.

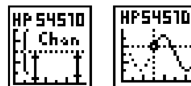
If your VI is a mathematical function, draw a plot of that function.



For simple data processing functions, depict the input and output data types and the nature of the operation. This can become cryptic, however, so be careful.

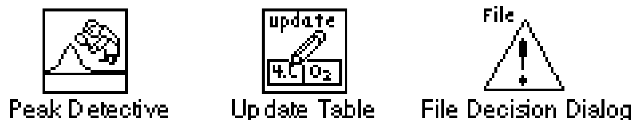


Within driver packages, maintain a unifying theme for groups of icons. Copy from drivers for similar instruments. This makes it easier for users to convert from one instrument to another with minimal confusion.



Do not spoil the international language of pictures by creating an icon that is a play on English words. For example, do not represent a data logging VI by a picture of a lumberjack.

Icons for higher-level VIs might require some artistic talent. Here are a few good icons, which are even better in color.



The Icon Editor features useful tools for creating icons. For example, to make symbols for the various inputs and outputs on the icon, you can display the connector pattern in the Icon Editor.

Use the Labeling tool to add text to an icon. Double-click on the Labeling tool to change the font or font size. Some fonts, like Symbol and Glyph, contain many small pictures you can use in your icons. Because it is in bitmap form, text typed in the Icon Editor does not change when viewed on a machine with different fonts.

The Block Diagram

The block diagram concept used in LabVIEW and BridgeVIEW is considered a breakthrough in software engineering. Like any new tool, developers still are learning optimal methods for its application. Fortunately, programming in LabVIEW and BridgeVIEW is graphical, hence the name of the language—G, so we can create programs that are both functional and visually engaging. This section contains recommendations for improving block diagrams—both in function and appearance.

Wiring Etiquette

Haphazard wiring can distract the user and make block diagrams difficult to follow. Align and distribute objects to make a block diagram as neat as the front panel. Employ symmetry and straight lines to make the block diagram easy to read. Do not hide objects behind structures or other objects.

The following are some general wiring tips:

- Never route wires through an icon to a terminal on the other side of the icon.
- Avoid routing wires underneath structures or icons.
- Do not use local variables just to avoid having long wires. Every local variable that reads the data makes a copy of it and also can lead to race conditions. See Chapter 26, *Performance Issues*, in the *G Programming Reference Manual*, or Chapter 27, *Performance Issues*, in the *LabVIEW User Manual* for more information.
- Reduce the number of pivot points in lines by aligning data sources and sinks. Use the cursor keys to remove single-pixel kinks from wires.
- Delete excess wires, such as loops, in wires.
- Evenly space parallel wires in straight lines and around corners.

Notice object alignment, consistent spacing, and labels on the long wires illustrated in Figure 7-4.

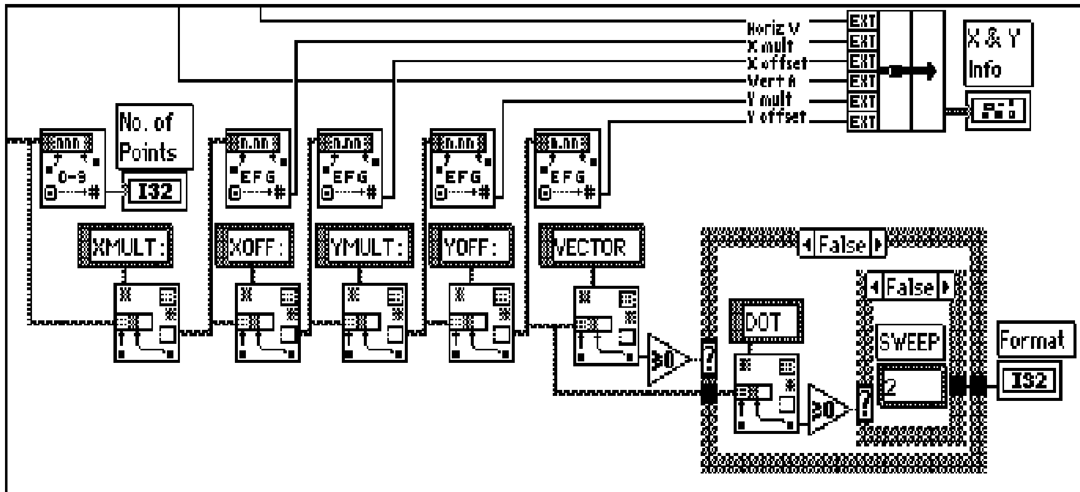


Figure 7-4. Good Wiring in a Simple Block Diagram

Labeling

Give major structures in the block diagram names and descriptions through the **Description** pop-up item. This helps the user understand complex segments of code.

Use enumerations as inputs to Case structures when possible, because the names from the enumerations appear at the top of Case structures instead of numbers. Add comments to explain the purpose of each frame. For comments, choose a font size and style that will stand out. Always label constants because they are not self-explanatory. Show the label of a subVI if the icon does not describe the function of the VI sufficiently.

Use free labels on long stretches of wire to label the signal data. Place the label right on top of the wire with a transparent border as shown below:

---------- signal name -----------

Paste long comments into small string constants and make them scrollable. Place large scrollable text items off to the side to avoid cluttering the screen.

When a VI is loaded on a different platform, the fonts change. Front panel labels might move automatically if they overlap controls. Block diagram elements do not move to accommodate font changes. Place labels below objects, so if they grow and shrink on the bottom and right sides, they will stay next to the object. Right-justify any labels you place to the left of an object.

Execution Sequence

The following sections describe G programming concepts that will help you take advantage of the natural dataflow in G.

Left-to-Right Layouts

G was designed to use a left-to-right (and sometimes top-to-bottom) layout. Your block diagrams should follow this convention. While the positions of program elements do not determine execution order, avoid wiring from right to left. Only data connections (wires) and structures determine execution order.

Data Dependency

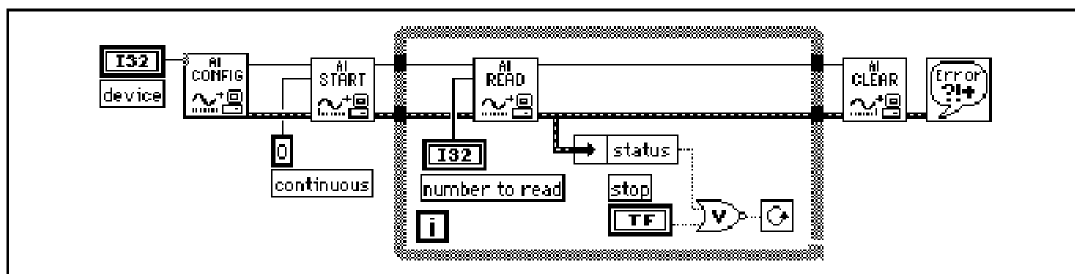
As described in Chapter 18, *Structures*, in the *G Programming Reference Manual*, or Chapter 19, *Structures*, in the *LabVIEW User Manual*, artificial data dependency should be applied wherever practical. If a section of the block diagram is missing the appropriate inputs or outputs, you might use a single-frame Sequence structure. Do not overdo it, though. To impose a pure dataflow model just for the sake of avoiding Sequence structures completely, is as bad as overusing them. Use dataflow programming techniques to create a clear, single-page main program.

Adding Common Threads

If you make a collection of subVIs that are used together often, give them all a common input/output terminal pair to chain them together without requiring Sequence or Case structures. A good example of a common thread is an error code. Each VI should test the incoming error and not execute if there is an existing error, then pass that error (or its own error) to the output. The only exception to this rule is a function like the Close File VI, which must perform cleanup regardless of whether an error occurred previously. The Close File VI closes the specified file and passes on error input as its output. This error information is commonly kept in a cluster containing a numeric error

code, a string containing the name of the function that generated the error, and an error Boolean for quick testing. This technique is sometimes called Error In/Error Out and is used in most of the I/O libraries.

The following figure shows an example of the error cluster used by data acquisition VIs. The While loop stops if an error is detected, and the General Error Handler VI reports the error to the user at the end. Notice the clean appearance of this style of programming.



Sequence Structures

Avoid overusing Sequence structures. G has a great deal of inherent parallelism. (Future computers with multiprocessor architectures could make good use of this feature.) Using a Sequence structure guarantees the order of execution but prohibits parallel operations. For instance, asynchronous tasks that use I/O devices (GPIB, serial, plug-in boards) can run concurrently with CPU-bound operations. Actually, your program might execute faster if you can add parallelism by reducing the use of Sequences. Sequence structures add *no code or execution overhead, but they do restrict parallelism*. Sequences hide parts of the program and interrupt the natural left-to-right flow of data.

While pure dataflow programming means avoiding Sequence structures, there are cases where it is appropriate to use them. Only use Sequence structures if one node must execute before another and cannot be connected by a wire. See the *Data Dependency* section earlier in this chapter for more information. Sequence structures also can be used to conserve screen space, although proper use of subVIs is better.

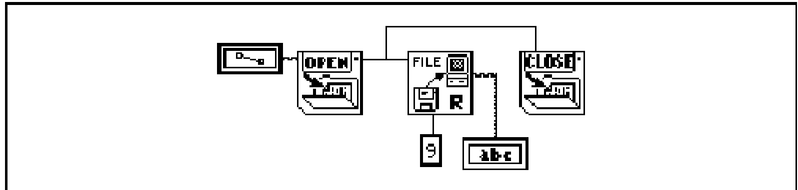
There is an alternative to the Sequence structure, as described in Lesson 8, *Additional Topics*, in the *LabVIEW Advanced Course Manual*, called a State Machine. Use a Case structure wired to a counter in a For or While loop. This technique allows you to jump around in the

sequence by manipulating the counter. For instance, any frame can jump directly to an error handling frame.

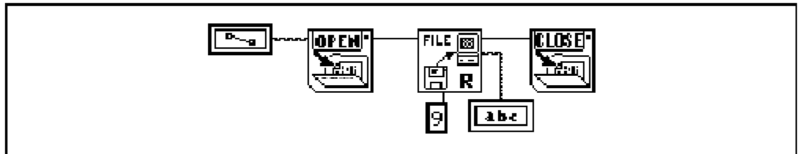
Watch Out for Missing Dependencies

Make sure that you have explicitly defined the sequence of events, when necessary. Do not assume left-to-right or top-to-bottom execution when no data dependency exists.

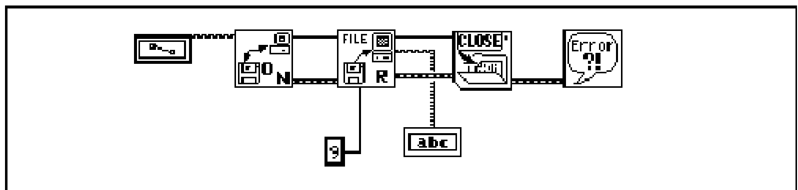
In the following example, there is no dependency between the Read File and Close File. More than likely, this program cannot work as expected.



The following version of the block diagram establishes a dependency by wiring an output of the Read File to the Close File; the operation cannot end until the Close File receives the output of the Read File.



Notice that the preceding example still does not check for errors. For instance, if the file does not exist, the program does not display a warning. The following version of the block diagram illustrates one method for handling this problem. In this example, the block diagram uses the error I/O inputs and outputs of these functions to propagate any errors to the simple error handler VI.



Check for Errors

When you perform any kind of I/O, consider the possibility of errors occurring. Almost all I/O functions return error information. Make sure that your program checks for errors and you handle them appropriately.

BridgeVIEW and LabVIEW do not handle errors automatically, because users usually want very specific error-handling methods. For example, if an I/O VI in your block diagram times out, you might or might not want your entire program to halt. You also might want the VI to retry for a certain period of time. In BridgeVIEW and LabVIEW, you make error-handling decisions.

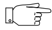
The following list describes three situations in which errors frequently occur:

- Incorrect initialization of communication or data that has been improperly written to your external device
- Loss of power, broken, or improperly working external device
- Bugs in BridgeVIEW, LabVIEW, or other programs that occur when you upgrade BridgeVIEW, LabVIEW, or your system operating software

When an error occurs, you might not want certain subsequent operations to take place. For instance, if an analog output operation fails because you specify the wrong device, you might not want BridgeVIEW or LabVIEW to perform a subsequent analog input operation.

One method for managing such a problem is to test for errors after every function, and put subsequent functions inside case structures. This can complicate your diagrams and ultimately hide the purpose of your application.

An alternative approach, which has been used successfully in a number of applications and many of the VI libraries, is to incorporate error handling in the subVIs that perform I/O. Each VI can have an error input and an error output. You can design the VI to check the error input to see if an error has previously occurred. If there is an error, the VI can be set up to halt execution and to pass the error input to the error output. If there is no error, the VI can execute the operation and pass the result to the error output.

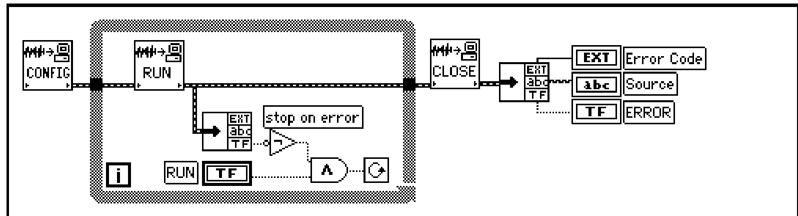


Note: ***In some cases, such as a* Close *operation, you may want the VI to perform the operation regardless of the error that is passed in to it.***

Using the preceding technique, you can easily wire several VIs together, connecting error inputs and outputs to propagate errors from one VI to the next. At the end of a series of VIs, you can use the Simple Error Handler VI to display a dialog box if an error occurs. The Simple Error Handler VI is located in the **Functions»Time & Dialog**. In addition to encapsulating error handling, you can use this technique to determine the order of several I/O operations.

One of the main advantages in using the error input and output clusters is that you can use them to control the execution order of dissimilar operations.

The error information is generally represented using a cluster containing a numeric error code, a string containing the name of the function that generated the error, and an error Boolean for quick testing. The following illustration shows how you can use this in your own applications. Notice that the While Loop stops if it detects an error.



Sizing and Positioning of Block Diagrams

Like the front panel, the block diagram should fit the user's monitor. The amount and density of wiring on a diagram often indicate the programmer's skill, forethought, and intentions. Having to remove a section of code and put it in a subVI because you ran out of space is a sign of not using effective top-down design.

When both the front panel and block diagram fit on an average-sized monitor, place the block diagram to the right of or below the front panel. This arrangement gives users the optimum view of your program. When both the block diagram and front panel do not fit on one screen, place block diagrams in the upper left-hand corner of the screen. If possible, show the title bar of both the front panel and block diagram. To set a window's position, move or modify any object and **Save** or select **Save As...** and use the same name.

Figure 7-5 illustrates how both a front panel and block diagram can fit comfortably on a small monitor with room to spare for the Help window.

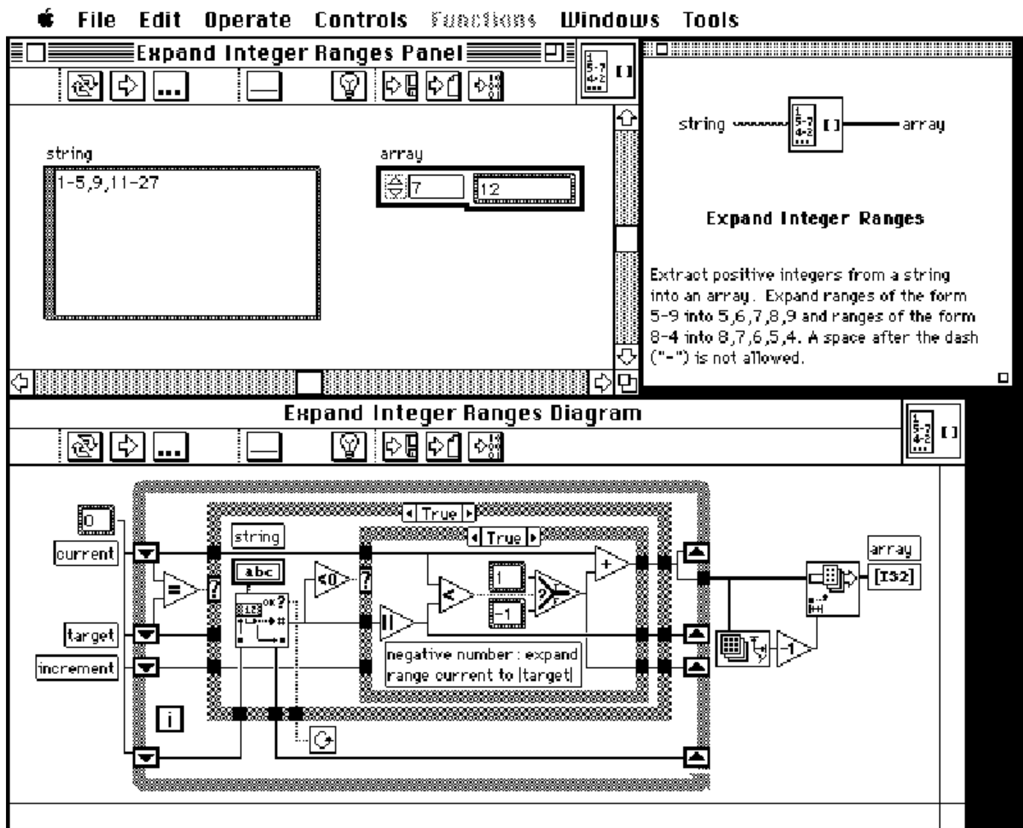


Figure 7-5. A Well-Placed Front Panel and Block Diagram

Optimization

There are many things you can do to optimize memory usage and execution time of your G program. Generally an advanced topic, optimization quickly becomes a concern when your program has large arrays and/or critical timing problems. See Chapter 26, *Performance Issues*, in the *G Programming Reference Manual*, or Chapter 27, *Performance Issues*, in the *LabVIEW User Manual* for more information on optimizing G programs.

Code Interface Nodes

Code Interface Nodes (CINs) can obscure the function of your VIs. Use CINs only when absolutely necessary. Include the information listed below to help your users understand what your CIN does and how to rebuild it.

CIN Description Contents

In the **Description...** pop up of a Code Interface Node, or in a scrolling label next to the node, record the following information:

- Source code file name
- Platform and operating system
- Compiler and version
- Location of the source code
- What the code does
- List of other files required to build the CIN
- Other critical information required to maintain the CIN

CIN Source Code

You should enter the same kind of information that is entered into the **Description...** pop up into the header file with the source code. If the source code is not too long, paste it into a scrollable block diagram string constant.

Style Checklist

Use the following checklist to help you maintain consistent style and quality. We recommend you copy this checklist for use on all of your projects.

VI Checklist

- ❑ Organize VIs in a hierarchical directory with easily accessible top-level VIs and subVIs in subdirectories.
- ❑ Avoid putting too many VIs in one library because large LLBs take longer to save.
- ❑ With LLBs, use Edit VI Library to mark top-level VIs.
- ❑ If the VIs will be used as subVIs, use Edit Control and Function palettes to create a .mnu file or edit the menu that is part of the LLB. Be sure to do the following:
  - Arrange palettes.
  - Name menus.
  - Hide dependent subVIs.
- ❑ Give VI meaningful names *without* special characters such as \, /, :, and ~.
- ❑ Use standard extensions so Windows and Unix can distinguish files (.vi, .ctl).
- ❑ Capitalize Initial Letters of VI Names.
- ❑ Distinguish example VIs, top-level VIs, subVIs, controls, and globals by saving them in subdirectories, separate libraries in the same directory, or by giving them descriptive names such as `MainX.vi`, `Example of X.vi`, `Global X.vi`, and `TypeDef X.ctl`.
- ❑ Write a VI description; proofread it; check Help window.
- ❑ Include your name and/or company, and the date in the VI description.

- ❑ When you modify a VI, use the history window to document your changes.
- ❑ Create a meaningful black & white icon (color icons optional).
- ❑ Make a connector pane; provide in and out data flow; leave extra inputs and outputs for later development; use consistent layout.
- ❑ Consider VI and window options carefully. Remember the following:
  - Do not set higher priority without serious thought.
  - Remember that hiding menu bars and using dialog box style makes Help and VI description inaccessible.
  - Hiding Abort and debugging buttons increases performance slightly.
- ❑ Set print options to print attractive output in the most useful format.
- ❑ Make test VIs that check error conditions, invalid values, and cancel buttons.
- ❑ Save test VIs in a separate directory so you can reuse them.
- ❑ Load and test VIs on multiple platforms, making sure labels fit and window size and position are correct.

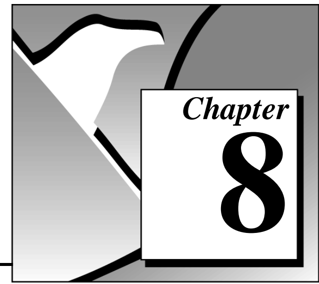
Front Panel Checklist

- ❑ Give controls meaningful names; use consistent capitalization, preferably all lower case or Initial Capitals.
- ❑ Make name label background transparent.
- ❑ Check for consistent placement of control names (e.g. upper left).
- ❑ Use standard, consistent fonts throughout all front panels.
- ❑ Use Size to Text for all text (for portability), add carriage returns if necessary.
- ❑ Use required, recommended, and optional settings on your connector pane.

- ❑ Put default values in parentheses after input names.
- ❑ Include unit information in names, if applicable, e.g. time limit (10 seconds).
- ❑ Write descriptions for controls (including array/cluster/refnum elements). Remember description might need to be changed if control is copied.
- ❑ Arrange controls logically—for top-level VIs, put the most important controls in most prominent positions; for subVIs, put inputs left, outputs right, and follow connector arrangement.
- ❑ Arrange controls attractively, using Align & Distribute.
- ❑ Do not overlap controls.
- ❑ Use color logically and sparingly, if at all.
- ❑ Use error in, error out clusters where appropriate.
- ❑ Consider other *common thread* controls, such as taskID, refnum, and name.
- ❑ Provide a **Stop** button if necessary; do not use the **Abort** button to stop a VI (Hide the **Abort** button).
- ❑ Use rings and enumerations where appropriate; if using a Boolean for two options, consider using an enumeration instead for future expansion of options.
- ❑ Use custom control or TypeDef for common controls (esp. for rings and enumerations); include it with VIs.
- ❑ In control VI, label controls with the same name as the VI (e.g., `Alarm Boolean.ctl` has default name Alarm Boolean).

Block Diagram Checklist

- ❑ Avoid creating extremely large block diagrams; limit them to one to two screens if possible.
- ❑ Label controls, important functions, and subVIs, constants, attribute nodes, locals, globals, and structures.
- ❑ Add comments (use object labels instead of free labels where applicable, scrollable strings for long comments).
- ❑ Make comment background transparent to distinguish from name labels.
- ❑ Place labels below objects when possible and right-justify text if label is placed to the left of an object.
- ❑ Use standard, consistent font conventions throughout.
- ❑ Use Size to Text for all text, add carriage returns if necessary.
- ❑ Reduce white space in smaller block diagrams, but allow at least 3–4 pixels between objects.
- ❑ Flow data from left-to-right; wires enter from left, exit to right (not top or bottom).
- ❑ Align & distribute functions, terminals, constants.
- ❑ Label long wires with small transparent labels.
- ❑ Do not wire behind objects.
- ❑ Make good use of reusable, testable subVIs.
- ❑ Make sure the program can handle error conditions and invalid values.
- ❑ Show name of source code, or include source code, for any CINs.
- ❑ Save with most important or first frame of structures showing.
- ❑ Review efficiency (especially data copying) and accuracy (especially parts without data dependency).



VI Metrics Tool

This chapter describes how to use the VI Metrics tool to measure the complextity of your application.

The VI Metrics tool provides a way to measure the complexity of an application similar to the widely used Source Lines of Code (SLOCs) metrics for textual languages. With the VI Metrics tool you can view statistics about VIs, which can be useful in finding areas that are too complex or in establishing baselines for estimating future projects.

To use the tool, first open the VI(s) that you want to analyze. Then select **VI Metrics** from the **Project** menu. The VI Metrics dialog box appears.

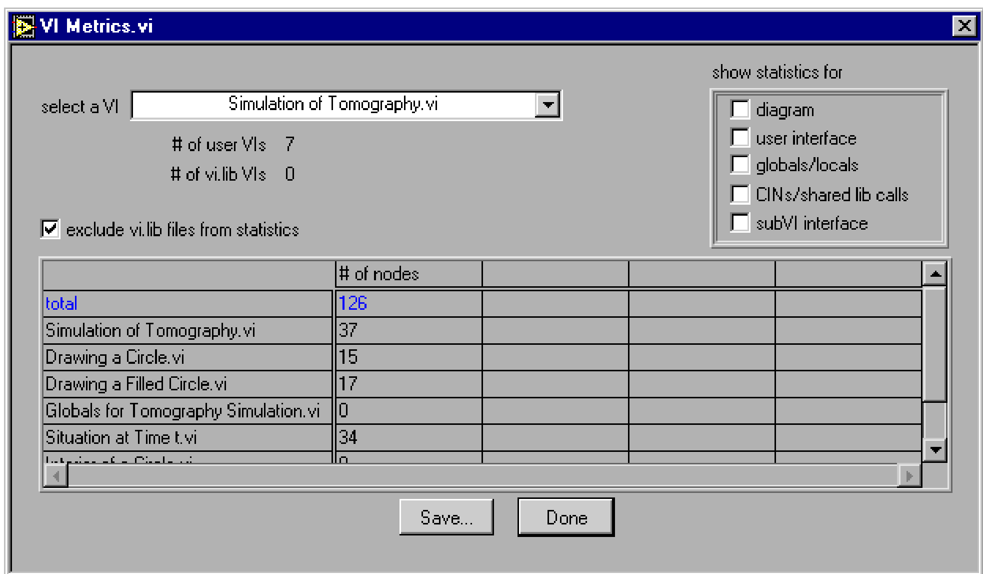


Figure 8-1. VI Metrics Tool Dialog Box

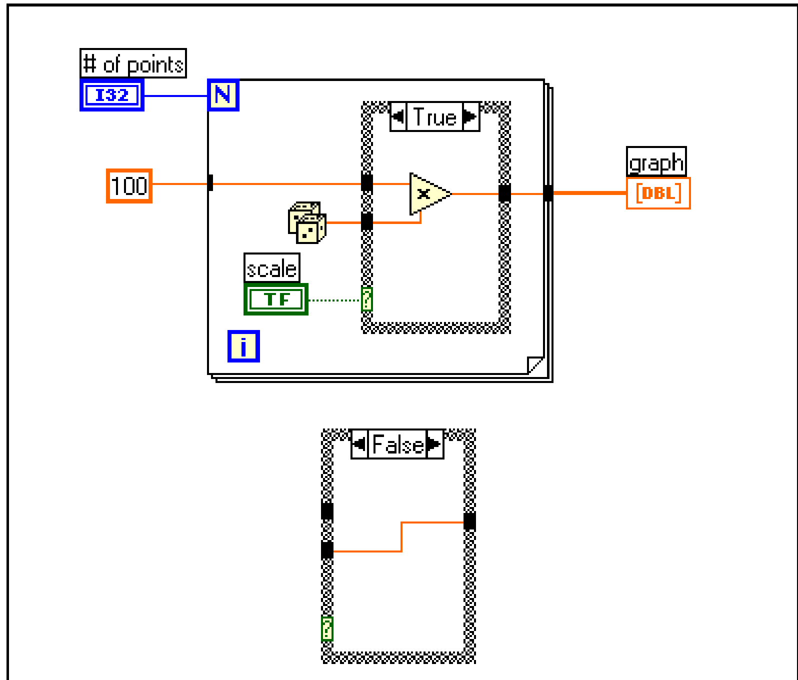
Use the ring at the top of the dialog box to select from the list of VIs with open front panels in memory. After you select a VI, the dialog box updates the list at the bottom with the names of the VI and its subVIs plus information on each VI.

For each VI in the selected hierarchy, the dialog box lists the number of nodes contained within that VI. Nodes are the executable objects on your block diagram. They are analogous to statements, operators, and subroutine calls in conventional programming languages. This number gives you a rough metric that is comparable to the Source Lines of Code metric commonly used with textual languages.

The number of nodes includes functions (Add, Subtract, and so on), subVI calls, and structures (case, while loop, and so on). It also includes terminals for front panel objects, constants, local and global references, and attribute nodes. Notice for attribute nodes, reading multiple attributes with the same node counts as one node. If you want to know the total number of attributes read or written by a VI, see the User Interface Statistics section later in this chapter.

The number of nodes does not include wires, tunnels, or objects that are subcomponents of structures such as the loop iteration count of a for loop or a sequence local.

As an example, the following block diagram contains eight nodes (three terminals, a constant, a random number function, a multiply, a case structure, and a for loop).



Additional Statistics

In addition to measuring the number of nodes, the dialog box also lets you measure a number of other statistics related to the complexity of your VI(s). To show the additional information, turn on the appropriate category checkbox at the top of the dialog box.

Block Diagram Statistics

- Structures—Number of for loops, while loops, case structures, and sequence structures.
- Block Diagrams—Number of block diagrams. Each VI has a single top level block diagram. In addition, it has one subdiagram for each loop and for each frame of a sequence or case.
- Maximum block diagram depth—Deepest nesting level of block diagrams in a VI. If your VI has no structures (cases, loops, sequences), it has a depth of 0.
- Diag width—Width of the block diagram in pixels.

- Diag height—Height of the block diagram in pixels.
- Wire sources—Wire sources measures the total number of sources in your VI. Each wire has a single source, but it can branch to multiple destinations. If, however, a wire crosses from one block diagram to another (through a tunnel), then the tunnel is considered to be a new source.

User Interface Statistics

- Controls—Number of top level controls on a VI front panel. A cluster or an array is counted as only a single control.
- Indicators—Number of top level indicators on a VI front panel. A cluster or an array is counted as only a single indicator.
- Attribute reads—Number of attribute reads by a VI block diagram. Note that if you read multiple attributes with the same attribute node, each attribute increments this number.
- Attribute writes—Number of attribute writes by a VI block diagram. Note that if you write multiple attributes with the same attribute node, each attribute increments this number.

Global/Local Statistics

- Global reads—Number of reads of global variables in a VI block diagram.
- Global writes—Number of writes to global variables in a VI block diagram.
- Local reads—Number of reads of local variables in a VI block diagram.
- Local writes—Number of writes to local variables in a VI block diagram.

CIN/Shared Library Statistics

- CINs—Number of Code Interface Nodes in a VI's block diagram.
- Shared library calls—Number of Call Library Nodes in a VI block diagram.

SubVI Interface Statistics

- Connector inputs—Number of controls on a VI connector pane.
- Connector outputs—Number of indicators on a VI connector pane.

Files in vi.lib

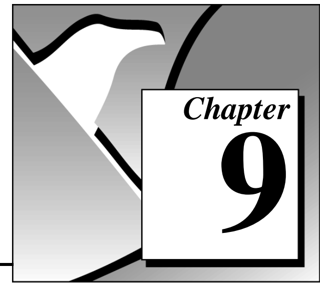
By default, the dialog box excludes VIs in vi.lib from the listing and from the totals. Calls to vi.lib VIs are counted as nodes, but information about the number of VIs that they call and the complexity of those vi.lib VIs are not added into the total measurements for the selected hierarchy. This is appropriate if you are trying to get a measurement of the complexity of the code that you have written. You can turn off the **exclude vi.lib files** option if you want to gather statistics on vi.lib VIs as well.

Saving Metric Information

You can save the metric information for a VI hierarchy to a text file using the **Save…** button. Only the columns that are displayed are saved. The information at the top of the dialog box concerning the number of user VIs and the number of library VIs is also saved to the file. The information is saved in a tab-delimited format so that you can easily read it into a spreadsheet or read and parse it using VIs.

As mentioned earlier, any metric, such as number of lines of code, is a crude measurement of complexity. This dialog box lets you access many statistics because you may find that some of the other columns are more valuable in some cases. For example, you might decide that, for user interface VIs, certain statistics can be combined to give you a better idea of how complex a VI is. In that case, you can make your own metric by saving the information about a VI and writing VIs to parse the results, combining fields to produce a new measurement of a VI complexity.

National Instruments is interested in hearing about combined or alternative metrics that you find useful in analyzing your VIs. You can use the Technical Support form at the back of the manual or our web site (`www.natinst.com`) to submit suggestions in this area.



Print Hierarchy Tool

This chapter descibes how the Print Hierarchy tool makes it easy to print out documentation for the VIs in your hierarchy. With this tool, you can automate the process of printing out VIs in any of the formats Print Documentation offers.

To use the tool, first open the VI(s) that you want to print. Then select **Print Hierarchy…** from the **Project** menu.

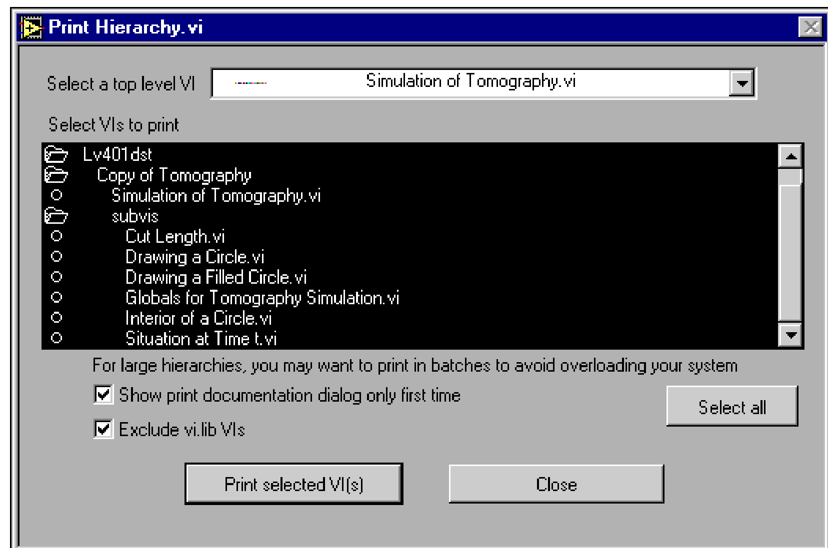


Figure 9-1. Print Hierarchy Tool Dialog Box

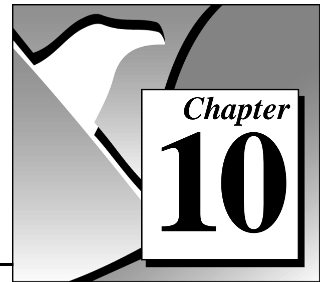
Using the ring at the top of this dialog box, select the name of the VI you want to print. The VI and its subVIs are listed in the listbox below this. By default, all files in the hierarchy are selected except VIs in vi.lib.

If you select **Print selected VI(s)**, the Print Documentation dialog box is displayed, giving you a chance to select the contents and layout of the printout. If you want to print VIs in vi.lib, turn off the **Exclude vi.lib VIs** checkbox.

By default, the tool prompts you only once for the format, and then uses that format information for all of the selected VIs. If you want to choose the format for each VI, turn off the **Show print documentation dialog only first time** checkbox at the bottom of the dialog box before printing the VIs.

For large hierarchies, you might not want to print the entire hierarchy at once. Because each printout is a separate print job, printing a large number of VIs can use a lot of space in the print queue. When printing many VIs, you might want to print them in sets of 10 to avoid filling up the queue.

File Manager Tool



This chapter describes how the File Manager tool makes it possible to easily copy, rename, or delete files within VI Libraries (LLBs). It also can be used to convert LLBs to directories, an important step if you want to manage your VIs with the Source Code Control tools.

To use the tool, select **File Manager…** from the **Project** menu.

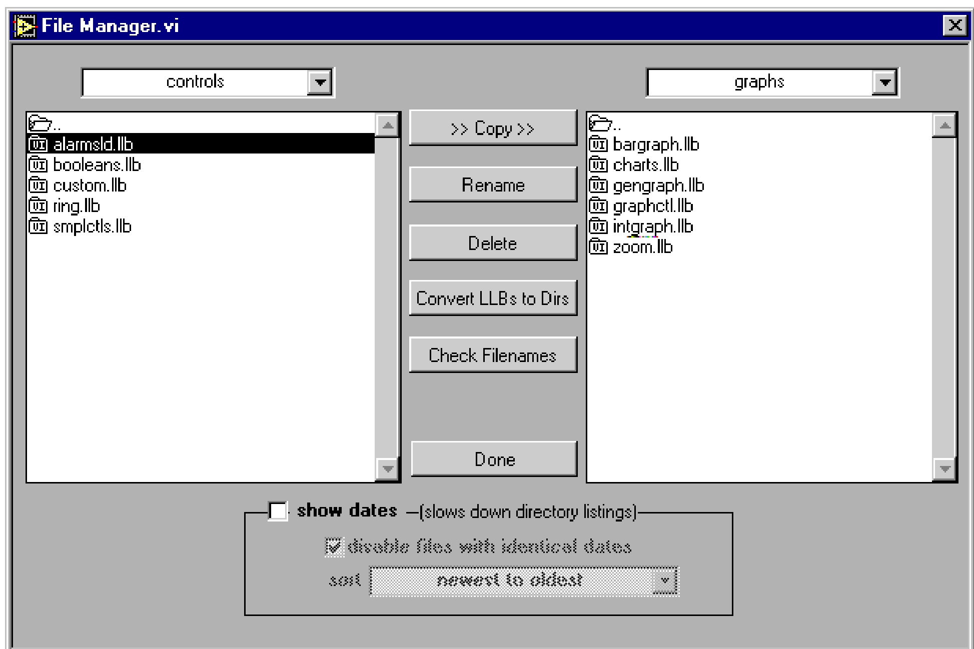


Figure 10-1. File Manager Tool Dialog Box

It is best to avoid performing a file operation on a VI that is already in memory, so you should close all VIs that might be affected before using this dialog box.

With this dialog box you can view two locations (directory or LLB) simultaneously. These listings behave like the normal file dialog box.

Once you have selected a file, you can copy, rename, or delete it using the corresponding buttons between the two lists.

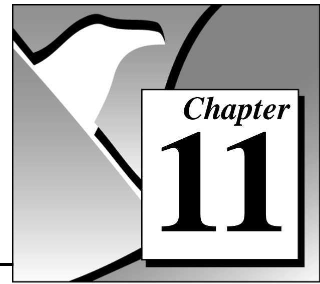
If you select an LLB, you can press the **Convert LLBs to Dirs** button to convert it to a directory of VIs. If you select a directory and press this button, the tool will scan for all LLBs within that directory and give you the option to convert them to directories. The new directory is created in the same location as the original LLB.

If you give the new directory a name that is different from that of the original LLB, LabVIEW or BridgeVIEW has to search for the files that were within the LLB when calling a VI (even when the name is the same minus the .llb extension). When you convert an LLB to a directory, you are given the option to back up the LLB (the `.llb` extension will be changed to `.llx`).

The **Check Filenames** button scans a directory or VI library for filenames that are not platform-independent. It scans all filenames to verify that they do not contain invalid characters (:, \, /, ?, *, <, >, or |). Filenames are also verified to be 31 characters or less (a limitation for the Macintosh). The **Check Filenames** option scans files within LLBs as well. Files within LLBs are portable, even if their names contain characters that are invalid on some platforms. This tool scans within LLBs to help you detect potential problems if you move your files out of VI libraries. Multiplatform filename issues are described in more detail in the *Multiplatform Issues* section of Chapter 11, *Source Code Control Tools*.

At the bottom of the dialog box are a set of options for displaying file modification dates next to each file. If you select this option, you can choose to sort the files by date and to gray out files that have the same name and date in both directory listings. This can be useful in comparing two directories to see if any files have changed.

Source Code Control Tools



This chapter describes the G Source Code Control tools. The G Source Code Control (SCC) tools, accessible from the **Project»Source Code Control** menu, let you add files to SCC and access those files from within the LabVIEW or BridgeVIEW environment.

General Source Code Control Concepts

Source Code Control tools help significantly in managing projects. They help with sharing files among multiple developers and multiple projects. SCC tools maintain a centralized master copy of project files. As you make changes, you update this master copy to reflect those changes. This makes it easy for any developer to access the latest version of the project files. Also, it encourages code reuse by making all code easily accessible.

SCC tools also help improve security and quality. When a developer decides to modify a file, he or she checks out the file, marking it so that other developers cannot modify the same file at the same time accidentally. When he or she completes his or her changes to the file, the developer checks in the new version of the file into Source Code Control, and it becomes part of the master copy of the project. If incorrect changes are made, most Source Code Control systems allow you to access previous versions of files.

SCC tools help track changes to your project. When a developer checks in a file, SCC tools ask the user to describe the changes. This information is maintained so that you clearly document the evolution of your project. In addition to maintaining the source code, SCC tools can manage all aspects of your project, including specifications and illustrations, and they can also keep track of the changes that are made to those documents. This ability to track the evolution of software is important to most organizations that are concerned with quality.

Using Individual Files Instead of VI Libraries (LLBs)

VI libraries (LLBs) give you a method of storing multiple VIs within the same file. The main advantage of this is that it allows you to create VIs with long, descriptive names even under Windows 3.1, in which filenames are limited to 8+3 characters in length. When you store VIs in a VI Library, only the VI library name itself needs to be limited to 8+3 characters in length.

You should not use VI libraries, however, for files that you want to put under Source Code Control. VI libraries are not practical for SCC because SCC tools cannot manage the files within a VI library individually. If the G SCC tools were to try to manage files within a VI library, you would have to check out the entire VI library when you want to make changes to any file within the VI library. Because VI libraries do not permit fine enough control over individual VIs, they are not supported by the G SCC tools.

The primary reason you might want to use VI libraries is that you need to support Windows 3.1, where filenames are very limited. The G Source Code Control tools are not supported under Windows 3.1. If you need to support Windows 3.1, consider whether you can do your main development under other supported operating systems such as Windows 95 or NT where you can use the G SCC tools. When doing development, save your VIs as individual files, not in VI libraries. When you need to release your VIs for use on Windows 3.1, you can save them into VI libraries at that point.

The File Manager tool described in the previous chapter can help in moving VIs to and from VI libraries.

Source Code Control Configuration

Before you can use the G Source Code Control tools, you need to install the software and configure it. When you install the software, you select whether you are installing as an administrator or a user. The administrator has an extra menu option: **Project»Source Code Control»Administration**. The administrator uses this option to set up the Source Code Control system so that it is usable by others. All users use the **Project»Source Code Control»Local Configuration** option to connect to the Source Code Control system and specify options such as where files are copied to locally from the Source Code Control system.

 © National Instruments Corporation

Selecting the Source Code Control System

The G Source Code Control tools are general-purpose because they are built on top of a generic plug-in architecture for managing Source Code Control. In addition to a built-in system for managing source code, they can communicate with third-party Source Code Control systems. At this time, Microsoft Visual SourceSafe for Windows 95 and NT is the only supported third-party Source Code Control system, but others can be added based on demand.

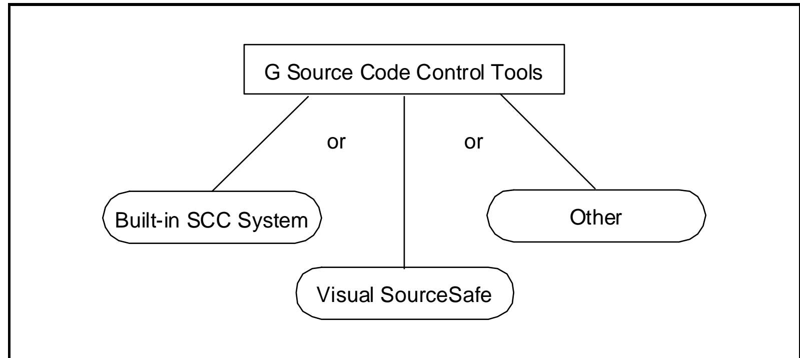


Figure 11-1. G Source Code Control Tools Work with Built-In and Third-Party Systems

Features of the Built-In Source Code Control System

The built-in Source Code Control System is a fairly simple system for sharing files. It manages files by storing them in a master directory (typically a network directory) that is accessible to everyone that needs access to the files.

The built-in system has many of the features found in full featured source code systems, including access to previous versions, history maintenance, and labeling of versions for easy retrieval.

Following is a list of some of the advantages the built-in system has over other third-party systems.

- The built-in system is available on all platforms for which LabVIEW and BridgeVIEW are available. Most third-party tools work only with one or two operating systems.
- The built-in system comes with the Professional G Developers Toolkit. Most third-party tools will add additional cost to deployment of a Source Code Control system within your organization.

There are some advantages, however, to using a third-party system in conjunction with the G SCC tools.

- A major difference between the built-in system and other systems is that while the built-in system provides source code access, it does not provide as much security as some other systems. Basically, the built- in system provides much of the full functionality of full Source Code Control systems, but it relies on trust more than other systems.

  It uses your LabVIEW or BridgeVIEW user name (**Edit»User Name...**) when checking files in and out. It does not require a password, so it can be circumvented if a user consciously wants to do so. Also, because the files are stored as normal files on a network accessible drive, theoretically users can modify the server files directly. Finally, some Source Code Control systems let you specify permissions for each file for each user so that some users might be able to read a file, but not modify it, while others have full access to a file. The built-in system does not currently let you specify file permissions.
- The built-in system does not use compression for files on the server and for storage of previous versions. Some Source Code Control systems store files on the server and differences between versions (deltas) in more efficient compressed formats. However, because VIs are stored in a binary format, most Source Code Control systems do not handle deltas on VIs very efficiently.

Features of Third-Party Source Code Control Systems

As mentioned earlier, the plug-in architecture of the G Source Code Control tools supports using third-party SCC systems. This plug-in architecture is based on VIs, so it is possible to even create your own if it is important to support a specific SCC system within your organization. At this time, Microsoft Visual SourceSafe for Windows 95/NT is the only supported third-party SCC system, but others may be added based on customer demand.

While you can use third-party tools, you might not be able to take advantage of all of the features that a given tool offers. For instance, the G Source Code Control tools do not support a feature found in some systems called *branching* where different projects use different versions of a VI. If you wish to branch development, you must do it by saving the VI with a new name.

Also, the G Source Code Control tools keep track of the status of files internally, including information on the projects a file belongs to, the user who has the file checked out, modification dates, and so on. Consequently, if you are using a third-party SCC system, you should not perform some operations “behind the back” of the G SCC tools. Specifically, if you want to check VIs in or out, you should do it using the G SCC tools. If you check a file in or out using the third-party tools directly (for example, from the SourceSafe environment), the G SCC tools will become confused.

Essentially, it is permissible to do any operation that does not modify the master copy of a file, including retrieving files, labeling files, and report generation. You also can use features of third-party tools such as file permissions to indicate that specific users cannot modify certain files. However, you should not modify the master file by deleting it or replacing it with a different version except from within the G SCC tools.

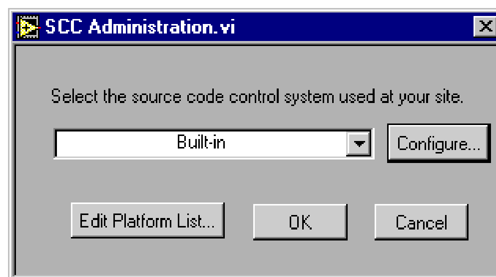
Microsoft Visual SourceSafe for Windows 95/NT

Visual SourceSafe is a popular Source Code Control package from Microsoft. It manages files on a network that is accessible to everyone who needs access to the files.

Visual SourceSafe manages the files and the deltas between files in its own internal database format and stores them in a compressed format that uses less storage. Also, it lets you specify user level permissions for specific files.

Administration (Administrator Only)

The Administrator has an extra menu option, **Project»Source Code Control»Administration**, that he or she uses to set up the SCC system so that it is usable by other users.



The primary thing that the Administrator does is select and configure the SCC system that the G SCC tools use for maintaining files. The ring lets him or her choose from the built-in system or other supported third-party systems (see the *Selecting the Source Code Control System* section for information on the differences between these options).

Administration of Visual SourceSafe

With Visual SourceSafe, it is important that you first install SourceSafe on a server and then use SourceSafe's administration tool to add user accounts for each user.

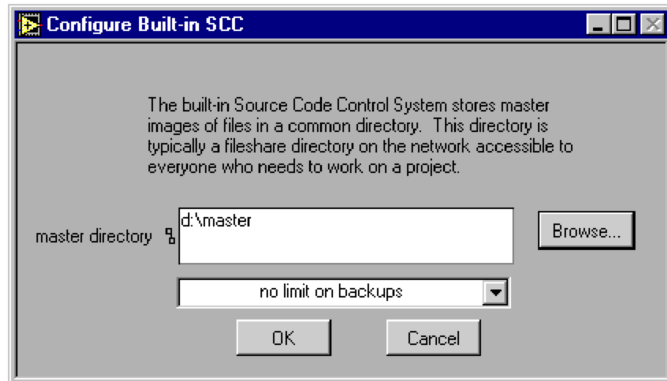
After you have installed SourceSafe, you need to run the **Project»Source Code Control»Administration** dialog box, select Microsoft Visual SourceSafe from the ring, and select OK to add configuration files used by the G SCC tools to SourceSafe's database. The G SCC tools maintain two files.

- `sccfiles.lst`—This is a master file list that the G SCC tools use to maintain information about each file, including the projects to which the file belongs.
- `sccplats.txt`—This file contains a list of the platforms that users can select for retrieving files.

If you want, you can also configure the platforms (that is, Windows 95, 68K Macintosh, and so on) that a user can select from this dialog box. See the *Edit Platform List* section for more information.

Administration of the Built-In System

If you select the built-in system, you are given a dialog box that lets you configure system-wide options that affect all users. If you need to change one of these settings later, you also can get to this dialog box by selecting **Configure…** from the **Administration** dialog box.

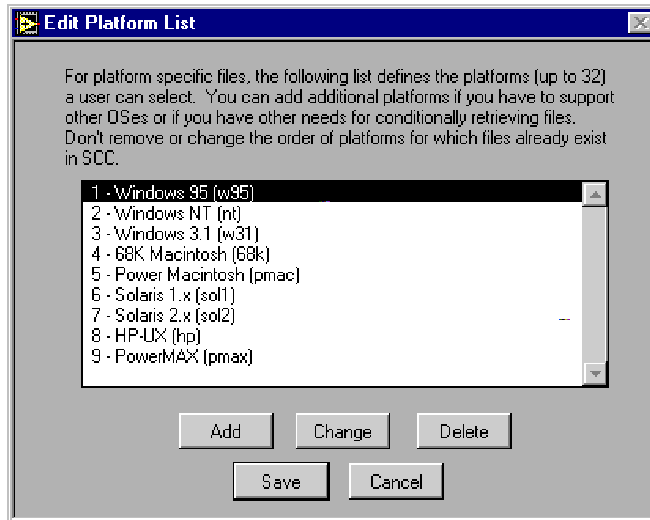


For the master directory, you should select a directory on a network file system that is accessible to all users. As users check files in and out, the files are copied to this directory, and history information for each file is maintained within this directory. Consequently, you should make sure that this directory has plenty of disk space available to it.

The ring at the bottom of this dialog box lets you configure how many backups should be maintained for files under Source Code Control. If you decide to maintain backups of files, when a user checks in a file, a copy of the old file is created. You can configure the system so that no backups are maintained, a specific number is maintained, or there is no limit on backups. In general, it is probably good to maintain a small number of backups so that you can retrieve older versions. If you choose to have no limits on backups, you will need to delete old versions periodically to avoid running out of storage space on the server. You can monitor the amount of storage used for backups from the **Project»Source Code Control»Advanced** dialog box.

Edit Platform List

Pressing the **Edit Platform List…** button in the Administration dialog box displays a list of the platforms that users can select from when they do local configuration.



Each entry consists of a long name and an abbreviation. The user sees long name when performing Local Configuration and if he or she chooses to mark a file as platform-specific. The abbreviation is used in file lists such as in the Advanced dialog box.

The main reason you might want to edit this list is if LabVIEW or BridgeVIEW becomes available for a new platform. In that case, you can add the name to the list and immediately be able to support it.

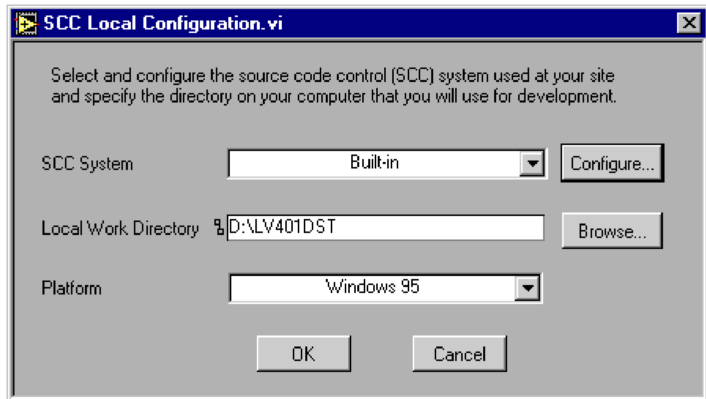
In general, it is probably a good idea to not modify this list, even if you initially only need to support a single platform. Assuming you do not change any of the existing items, the G SCC tools automatically detect the platform being used. If you delete one of the existing platforms and you later decide that you want to add support for that platform then you must add that name back in exactly as it was spelled originally to have the auto detection work correctly.

The list is limited to 32 entries. Do not change the order after you have added files, because each file remembers the platforms it applies to by number(s), not by name.

While Windows 3.1 is not supported directly by this toolkit, it is listed as an option so that you can have files specific to Windows 3.1. See the *Multiplatform Issues* section for more details.

Local Configuration (All Users)

Each user has to configure the G SCC tools before they can use them. You do this configuration using the **Project»Source Code Control»Local Configuration** dialog box.



The main thing each user needs to do in this dialog box is select the SCC system that the administrator configured. Ask your SCC administrator to determine which system your site uses.

User Configuration of Visual SourceSafe

To use SourceSafe, there is relatively little configuration needed. The primary thing you need to do is to install SourceSafe and make sure that your administrator has assigned you an account and password.

Other than that, just specify the local work directory and the platform, as described in the *Work Directory and Platform Configuration* section later in this chapter.

User Configuration of the Built-In System

Your system administrator should have configured a master directory on a drive or a network which is used for storing files under SCC. Users need to connect to that network drive so it is accessible to them for reading and writing.

From the **Project»Source Code Control»Local Configuration** dialog box, users should select the Built-in system. This displays a new dialog box that allows you to select the master directory. The dialog box verifies the path you specify is valid and has been set up by the Administrator for Source Code Control. You also can access this dialog box by pressing the **Configure...** button.

Work Directory and Platform Configuration

The local work directory is the directory in which all work you do is stored. The idea of the work directory is that all users will end up with the same set of subdirectories and files within that directory. As you change files within your directory and check them into Source Code Control, other users can copy the files from Source Code Control to their work directory.

The exact location for the work directory is completely up to you. You should make sure that you have enough disk space on the drive containing that directory, because it needs to be able to contain all files that you work on that are under Source Code Control.

The **Platform** ring lets you select the platform that you are working with. Unless your administrator has changed the setup, it should correctly autodetect the platform you are using. In general you will not need to change this setting. See the *Multiplatform Issues* section for information on the how the platform information is used and a description of situations when you might want to change it.

Managing Source Code Control Projects

Source Code Control Projects Overview

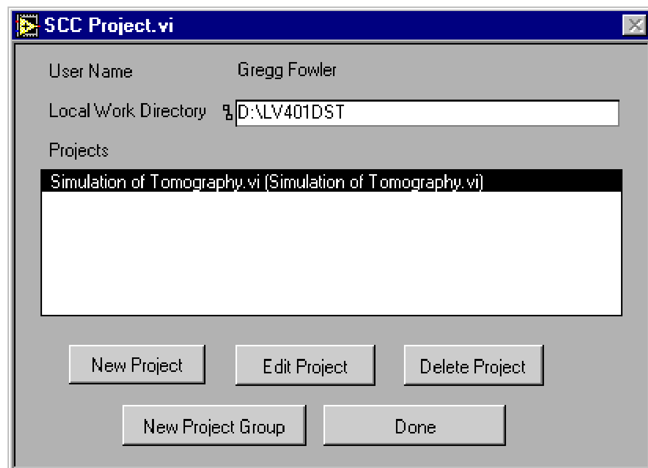
The G Source Code Control tools help you create projects under Source Code Control. A project is primarily a single VI hierarchy (VI plus all or some of its subVIs). In addition, a SCC project also can contain extra project related files such as specifications, shared libraries, and external subroutines.

The G SCC tools can maintain multiple projects. If you create two projects that contain the same subVI, only one copy of that subVI is maintained on the server.

By creating a project based upon a VI hierarchy, the G SCC tools can help you to keep the SCC project up to date. As you add files to or remove files from your hierarchy, the G SCC tools notice the changes and help you to update the project.

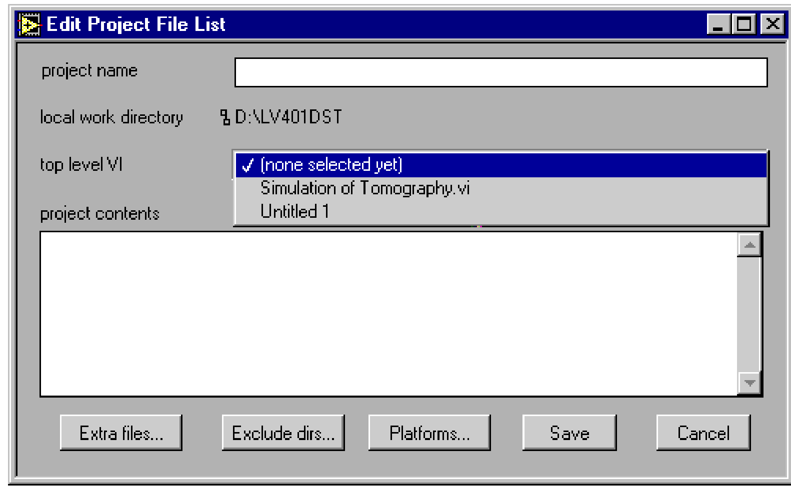
Some applications you develop might have more than one hierarchy. For example, if you use the VI control VIs to dynamically load a VI, you have one hierarchy for the main set of VIs and another for the VI you load dynamically. You should create separate projects for each hierarchy and then you can create a project group, as described in the *Project Groups* section, to make it more convenient to access the VI hierarchies simultaneously.

You create and edit the contents of projects using the **Project»Source Code Control»Project** dialog box.



Creating a Project

A Source Code Control project consists of a hierarchy of VIs (VI plus all or some of its subVIs). To create a project, open the top level VI that you want to add to SCC. Then, open the **Project»Source Code Control»Project** dialog box and select the **New Project** button. From the subsequent dialog box, select the top level VI for which you want to create a project. The project contents listbox updates the list of subVIs that the VI calls.



If you select **Save**, the project is created and all VIs in the listbox are added to SCC if they are not already part of SCC. As files are added to Source Code Control, they are locked to prevent accidental modification. When you want to modify a file that is under SCC, you must check out the file (see the *Checking Out Files* section later in this chapter).

If any files in your hierarchy are not present in your working directory or are in an LLB, those files cannot be added to Source Code Control.

With the **Extra Files** button, you can add files that are not part of your hierarchy to the project. You can use this to add project-related documents such as proposals, specifications, and illustrations to SCC. You also can use this to add support files such as shared libraries (DLLs) and external subroutines (`.lsb` files) that are not detected normally as part of your VI hierarchy, but are necessary to run your software. See the *Adding Extra Files to a Project* section for more information on this option.

The **Platforms** and **Exclude Dirs** buttons are advanced options that you typically do not need to modify.

The **Platforms** button lets you work with platform-specific files. If your application calls CINs or shared libraries (DLLs), you may end up with files that are specific to a given operating system. With a DLL, it might be available under Windows but not on other operating systems. If you write a CIN for multiple platforms, you will find that you need a different variant of the VI containing the CIN for each platform. The Platforms dialog box addresses both issues by letting you mark a file as platform-specific and also letting you create variants of a file for different platforms. See the *Multiplatform Issues* section for more information on this option.

The **Exclude Dirs** button lets you edit a list of directories that should be ignored as far as source code is concerned. For example, by default the files in the vi.lib directory are not listed as candidates for SCC. In general, you will not need to change this setting, although you may want to add a directory of your own if you have specific files that don't need to be added to Source Code Control.

Updating a Project

As you develop your VIs, you will create new subVIs and remove calls to subVIs. The integrated SCC tools make it easy to keep the SCC project up to date.

To update the SCC project, first open the top level VI for your project. Then, open the **Project»Source Code Control»Project** dialog box, select the project and select the **Edit Project** button.

If any VIs have been added or removed from your VI hierarchy, the **Edit Project** dialog box will prompt you whether you want to update the project to reflect those changes. As always, the files must be in your working directory and cannot be in LLBs unless they are in one of the exclude directories for the project (by default, files in vi.lib are excluded).

Removing Files from a Project

To remove files from a project, open the **Project»Source Code Control»Project** dialog box, select the project and select the **Edit Project** button. Then select the file(s) and press the **Remove** button or double click on the file.

When you remove VIs from a project, they remain listed in the project file list but they have an X next to them to indicate that they have been removed. You can add the files back to the project by double clicking on the file again (a check will appear next to the file).

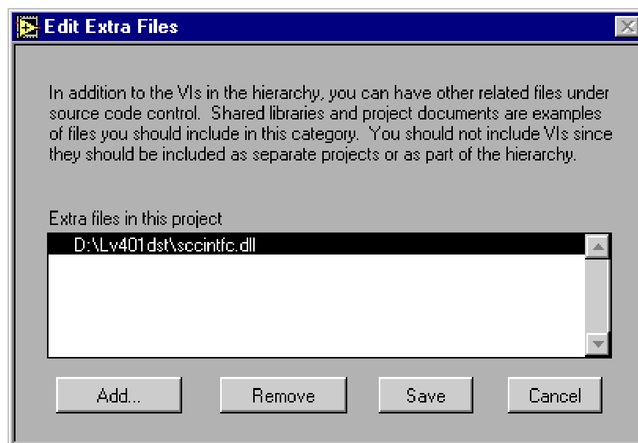
Another way to remove files from the project is to use the **Exclude Directories** dialog box and select the directory that contains the files as a directory to exclude. Files that are excluded in this fashion are grayed out in the project file list.

Notice that when you remove a file from a project, it is not removed from Source Code Control. One reason that they are not removed is that files can be shared by multiple projects. Also, even files that are not part of a project are retained in SCC because they may be important for historical reasons. To permanently delete a file, use the **Delete File** option from the **Project»Source Code Control»Advanced** dialog box.

Adding Extra Files to a Project

In many cases, the work you develop consists of more than just VIs. You probably have specifications, proposals, and illustrations that describe your project. You also can have support files like shared libraries (DLLs) or external subroutines. The integrated SCC tools support storing these extra, project related files as part of your SCC projects.

To add extra files to your project, open the **Project»Source Code Control»Project** dialog box, select the project and select the **Edit Project** button. Then press the **Extra Files** button.



Add or remove extra files by pressing the **Add…** or **Remove** button. Once you are finished, press **Save** to commit any changes you make.

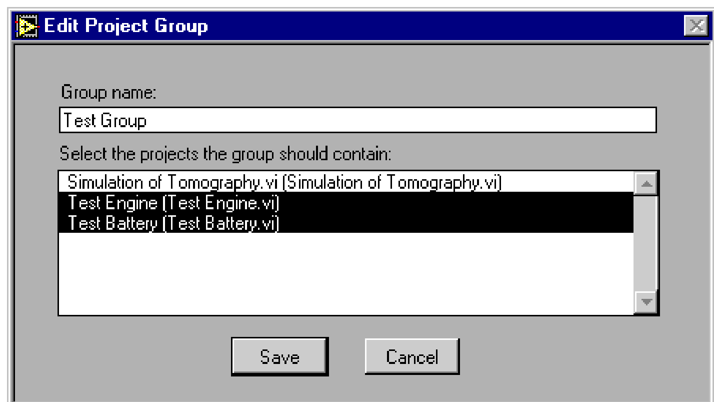
Do not use the Edit Extra Files dialog box to add VIs to a project. Instead, VIs should be added automatically if they are part of a hierarchy. If you have a set of VIs that are not part of the project but that you want to store in Source Code Control, create new project(s) for the additional VIs. If you want to be able to retrieve multiple projects simultaneously as though they were a single project, create a Project Group.

Project Groups

Each SCC project consists of a VI, all or some of its subVIs, and extra files associated with the project. Some development efforts you work on may consist of more than one top level VI. For example, if your VI uses the VI Control VIs to dynamically load and call VIs, the dynamically called subVIs are not really a part of your hierarchy. In this case, create separate projects for each dynamically called VI.

An SCC project group is a collection of projects. You can use project groups to make it more convenient to retrieve and manipulate files from multiple projects.

To create a project group, first define the individual projects that you want it to contain. Then select **Project»Source Code Control»Project** and press the **New Project Group** button.



The dialog box lets you enter a name for the group and select the projects that the group contains. Project groups can contain any number of projects and can contain references to other project groups.

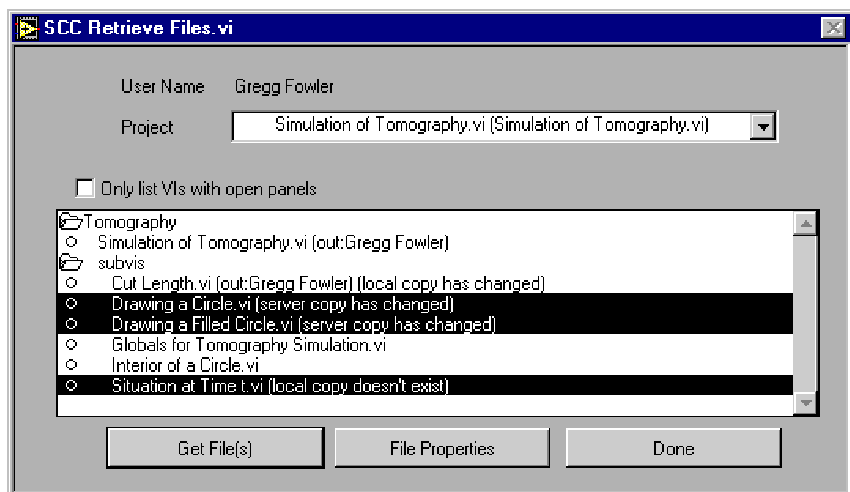
Once you have created a project group, its name shows up as a project you can select when you use the Checkin, Checkout, Retrieve, or Advanced dialog boxes.

Accessing Files

Retrieving Files

To copy files from the master directory to your working directory, select **Project»Source Code Control»Retrieve Files**.

The dialog box lets you select a project (or project group) from which to retrieve files.



To retrieve a file, select it and press **Get File(s)**. To retrieve multiple files, shift click on items to select them and then press **Get File(s)**. Since the list can get long if you have a lot of files in your project, you can use the **Only list VIs with open panels** checkbox to filter out unopened files.

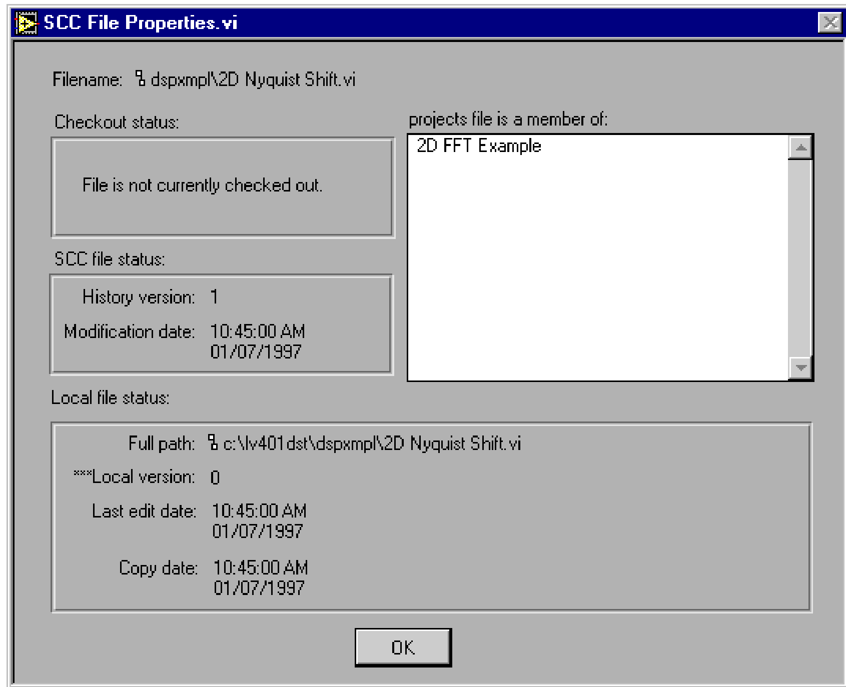
File Status

The dialog box automatically compares the local copy of files in the project with the master copy that the SCC system maintains. It categorizes the files into the following categories and indicates this information in parentheses next to each file in the list.

- Local copy has changed—If the file is not checked out, you should either check the file out or replace the local copy with the version from the server. It is a bad practice to modify local files without first checking them out.
- Server copy has changed—This generally means that another developer has modified the VI.
- Local copy does not exist—Either the file is a file on the server that you have never retrieved or you deleted it from your local system. If you have intentionally deleted it and you want to update the project, make sure to use the **Project»Source Code Control»Project** dialog box to update the project file list.
- Server copy does not exist—Either the file has been deleted on the server by another developer or it is a new file that you have created. If it is a new file that you created and you want to update the project, make sure to use the **Project»Source Code Control»Project** dialog box to update the project file list.

File Properties

The File Properties button displays a dialog box that gives you a summary of information about the file, including the projects it belongs to, the checkout status, and modification dates.



You can get more information about a file, including a file history, from the **Project»Source Code Control»Advanced** dialog box.

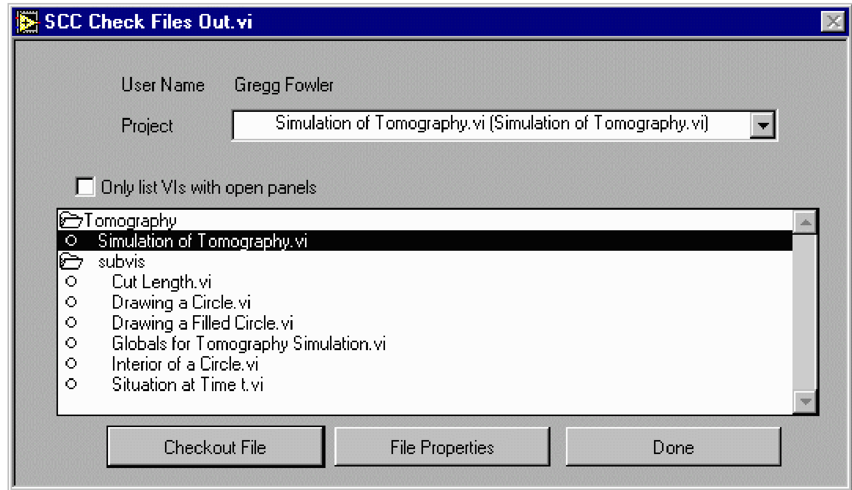
Checking Out Files

When you want to modify a file, first check the file out, which reserves it so that nobody else can modify the file.

When you check out a file, if there is a newer version on the server, that version is copied down to your local system. It is then unlocked so that you can edit the VI. While you have the file checked out, nobody else can check the file out or modify it. This helps to ensure that only one developer modifies a VI at a time.

To check files out, select **Project»Source Code Control»Check Out Files**.

The dialog box lets you select a project (or project group) from which to check files out. When you check a file out, it is checked out for all projects. When you subsequently check the file back in, the new version is available for all projects that contain the file.



The interface for checking files out is almost identical to that for retrieving files. You cannot check files out that are already checked out to either you or another developer. If it is checked out, the list indicates the user name of the developer with the VI. If you need to check multiple files out, shift click on items in the listbox to select the files you want to work on and press Checkout File(s).

In general, it is a good idea to avoid checking files out for long periods of time. Instead, you should try to make incremental changes to your files, and then when you are sure that they are in good shape, you should check them in. Whenever you check a file in, make sure that you have tested it well enough that you know that you are not going to cause problems for other developers. If you need to modify other VIs before you can check a specific VI in, check out the other VIs, make all of the changes, and test the VIs before checking any of the VIs in.

If you need to make several changes to a VI, consider checking the file in between modifications and then check the file back out to start the next modification. Not only does this allow other developers access to you changes, but it also gives you a checkpoint that you can return to if you later decide that your subsequent changes were incorrect.

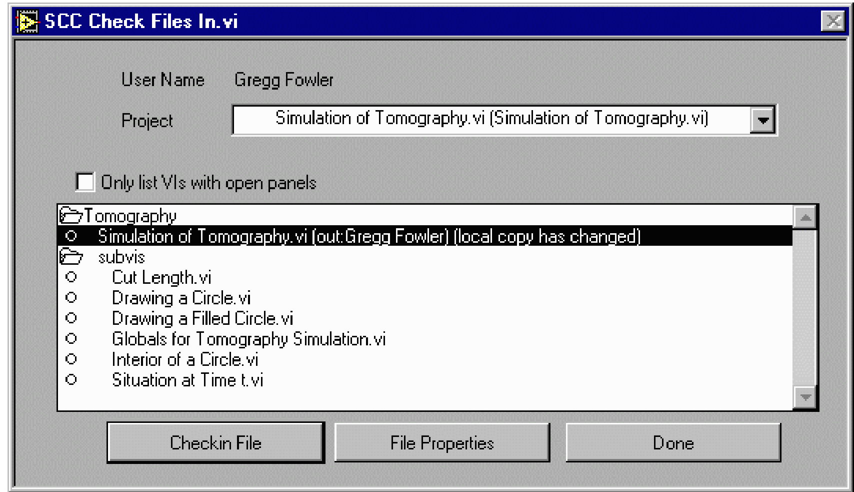
Use the History Window to Document Changes

As you make modifications, it is a good idea to use the History window (accessible from the **Windows** menu for a VI) to make notes about your changes. You can check a file out for several days, so the history window helps you to remember the changes you have made. When you check the file back in, the SCC tools let you enter a description of your changes. By default, this text is the history text since you checked the file out.

The more detailed you are in making notes, the better off you will be. These notes can help if you need to make reports about the changes you have made or if you later need to retrieve an old version and you need to distinguish between two different versions. You can create reports and access old versions using the **Project»Source Code Control»Advanced** dialog box.

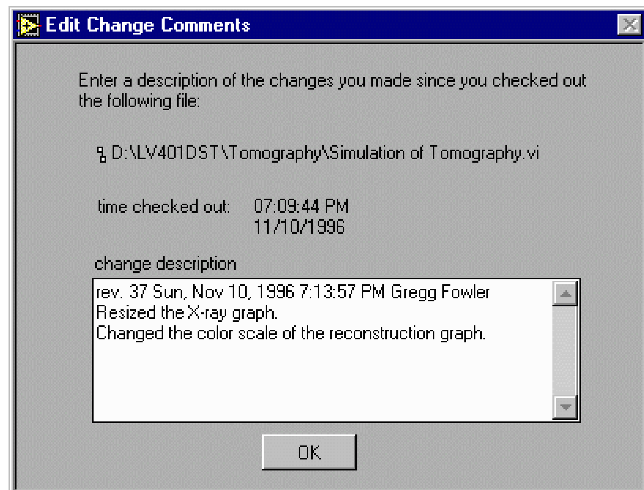
Checking In Files

When you are finished making changes to a file that you have checked out already, you can use **Project»Source Code Control»Check Files In** to copy your version into Source Code Control so that it is available to other users. The VI is locked automatically as it is checked in to prevent you or others from accidentally modifying the file without first checking it out.



The interface for checking files in is very similar to that for checking files out and for retrieving files. You only can check in a file if it is checked out to you (your user name must be the same as when you checked out the file).

When you check in a file, you are prompted to enter a summary of the changes you have made. If you have used the history window to record changes, this dialog box initially contains the history text since you checked the file out.



You can edit this text if you want to either expand upon the information or remove unimportant information. When you change this text, you are not modifying the VI history. Instead, the modified text is stored in Source Code Control with the file as part of a log of changes. This log is useful for report generation and can be helpful if you later need to retrieve an old version and you need to distinguish between two different versions. You can create reports and access old versions using the **Project»Source Code Control»Advanced** dialog box.

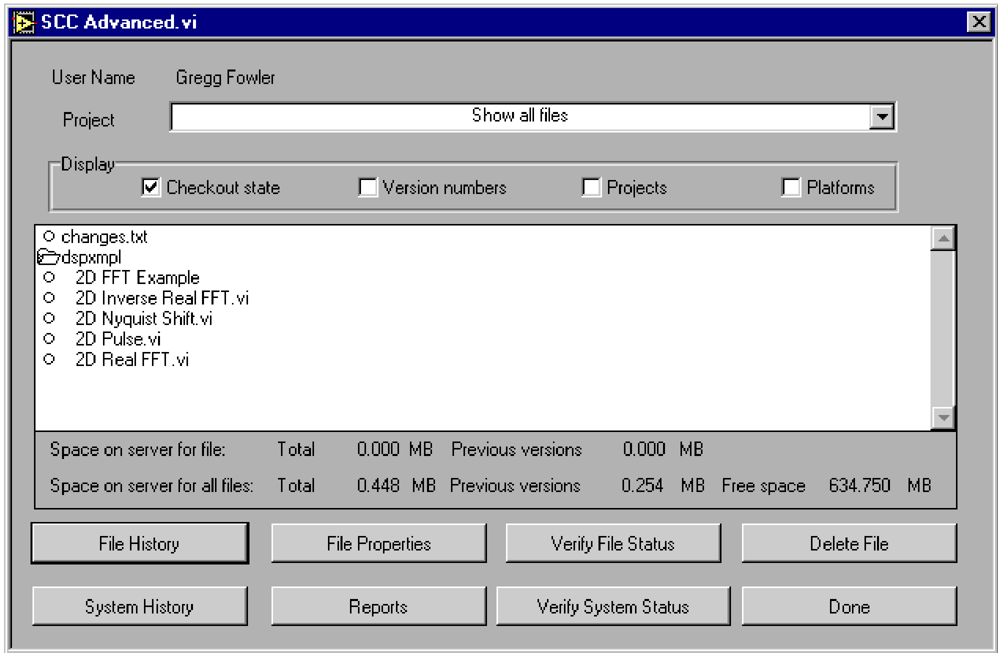
SCC User Name

When you check files out or in or modify projects, the G SCC tools use your LabVIEW or BridgeVIEW user name to access the Source Code Control system. This user name can be changed using **Edit»User Name**. Also, preferences in the **Edit»Preferences** dialog box under the History page let you control whether to prompt for a User Name when you launch LabVIEW or BridgeVIEW.

It is important that the user name be unique among your team. The built-in SCC system does not check for a password, so it is possible for users to check files in or out as another developer if you do not use unique names. The built-in system relies on a certain degree of trust in this area. If more security is important, consider using an alternative SCC system such as SourceSafe for storing files. With SourceSafe and other third-party tools you are prompted to enter a password whenever you access files or modify projects. See the *Source Code Control Configuration* topic for more information on the built in system and third-party systems.

Advanced Features

Most advanced features are accessed from the **Project»Source Code Control»Advanced** dialog box. This dialog box contains features for viewing all files under SCC and determining the projects those files belong to, accessing older versions of files, permanently deleting files, and creating reports.



Deleting Files from SCC

When you remove a file from a project, it is not removed automatically from Source Code Control. The main reason for this is that while a file may not be needed currently, it is important historically in terms of being able to understand the evolution of your software. Thus, if you later decide that you need to retrieve an older version of the project, the file is still present in SCC.

If you decide that you do not need a file, you can delete it using the **Project»Source Code Control»Advanced** dialog box. From the dialog box, select the files you want to delete and press the **Delete File** button.

Use caution in deleting files. In addition to removing the current version, it also deletes all previous versions of the file that the SCC system maintains and the history log for that file. Also, it is a good idea to enable the **Display Projects** option in the advanced dialog box to verify that the files you are deleting are not used currently by any projects before you delete the file.

SCC File History

Every Source Code Control transaction that modifies the contents of the SCC system is recorded. When you create projects, add files to projects, check files in and out, or delete files or projects, that information is logged, and the comments you enter when you check a file in are added to the log for a file.

To view the SCC history information for a file, use the **File History** button in the **Project»Source Code Control»Advanced** dialog box. It displays a scrolling list of the SCC history for the file, including dates and names for every person that modified the file.

It is important to understand the difference between the SCC File History and the VI history that you modify with a VI history window. The VI history window can be used as a place to write notes about changes to a VI as you make them. This VI history is a part of the VI, so if you give the VI to someone, the VI history is still present (unless you reset it using an option from the history window). When you check a file in, regardless of whether it is a VI or some other document such as a specification, you are given a chance to make an entry in the file's SCC file history. For VIs, the default text for this entry is the VI history text since you checked the file out. However, you are given a chance to change this entry so that the SCC history entry is more detailed or more succinct. This SCC file history information is maintained within the SCC system and does not become a part of the VI.

The dialog box appearance is dependent upon the Source Code Control system you use (either the built-in system or a third-party source code system). At a minimum, it lets you scroll through a listing of information about each file. In addition, for most systems it should allow you to access previous versions of files (see *Accessing Previous Versions of Files* for more information on this feature). Also, the built-in system (and some third-party systems such as SourceSafe) gives you the option of labeling the current version of a file so that it is easier to recognize if you later want to retrieve it; for example, you might label a file as beta so that you can easily retrieve that checkpoint version later on. See the *Labeling Versions of Files for Easy Retrieval* section for more information.

It is important that you be detailed in entering comments when you check files in (and if you use the history window) because that information can help you to subsequently understand the evolution of

your software and also is extremely helpful if you need to determine in which version a problem was introduced.

If you want to view a summary of transactions for multiple files, an easy way is to use the **System History** button instead of the **File History** button (see the *System History* section for more details). Also, if you use the report generation feature of the Advanced dialog box, you can save the System and File History for files in a Project (see the *Creating Reports* section for information on the options available in this area).

System History

The System History dialog box lets you view a brief summary of transactions affecting the Source Code Control system. It lists any transaction that modified the contents of projects, created or deleted files, or checked files in.

As with the File History dialog box, the System History dialog box appearance depends upon the Source Code Control system you use (either the built-in system or a third-party system). At a minimum, it lets you scroll through a listing of transactions. In addition, for most systems it allows you to label the current version of all files in projects so that they can be easily retrieved later on; for example, you might apply a label of *beta* to all files in a project (or multiple projects) so that you can easily retrieve those versions later on. See the *Labeling Versions of Files for Easy Retrieval* section for more information.

Master File List (sccfiles.lst)

If you are using a third-party system, it is helpful to understand how the system history information is maintained. The G Source Code Control tools maintain a binary file named `sccfiles.lst` that describes all of the projects and files in Source Code Control. Every time you check in or check out a file, or you modify a project, this file is modified. As with all other files in SCC, the `sccfiles.lst` file has history. However, rather than prompting you for comments or using the full comments you apply to each file, the G SCC tools automatically apply a brief comment for each transaction you make that describes the files or projects that were modified. For projects, it indicates in which way they were modified (files added or removed).

If you are using a third-party SCC system, System History lets the third-party SCC system display the history for this `sccfiles.lst` in the manner in which it normally displays history information.

Consequently, the dialog box you see might give you options for retrieving older versions of the sccfiles.lst or for labeling the file. You should not perform these operations on this file. If you accidentally modify the version of this file on the sever, the G SCC tools can become confused about the files under Source Code Control and the checkout state of those files.

Accessing Previous Versions of Files

A useful feature of most SCC systems is that they automatically maintain previous versions of files. This is helpful if you ever make a mistake and need to recall an older version of a file. It is also useful if you give a version to a customer, continue development, and subsequently need to retrieve the same version you sent the customer so that you can reproduce the system he or she has.

Built-In System

The built-in SCC system supports maintaining previous versions of files. The System Administrator can enable this feature, but he or she may choose to disable it for disk storage reasons (maintaining older versions of files can dramatically increase your storage requirements). In addition, the administrator can configure it so that it only maintains a limited number of previous versions of each file. In this case, as you check a newer version of a file in, a fixed number of previous versions will be maintained.

With the built-in system, if you label versions of files, the labeled versions are not automatically deleted and are not counted as part of this System Administrator limit. Labeled versions are maintained until such time as you choose to delete them.

With the built-in system, you can access previous versions of files from the File History and the System History dialog boxes. The File History dialog box allows you to retrieve a previous version of a single file. The System History dialog box allows you to scan the system for all versions of a files with a specific label and it allows you to retrieve those versions. This can be useful in taking a snapshot of your product that you can subsequently retrieve very easily (for example you can label the current version of all files in a project as `rel1` for the first release).

Third-Party Systems

Most third-party systems including SourceSafe offer similar features for maintaining previous versions of files and labeling files. The option might be configurable; consult the documentation for the third-party SCC system to determine the options. By default, SourceSafe maintains history for files.

Labeling Versions of Files for Easy Retrieval

With the built-in SCC system, the **File History** button in the **Project»Source Code Control»Advanced** dialog box lets you retrieve older versions of files. By default, however, the names you see are based upon the older version number and the version creation date. The File History dialog box lets you optionally label the current version so that it is easier to recognize if you later need to retrieve it. Also, labeling a file ensures that it is not automatically deleted as it gets older (one of the Administrator options for the built-in system is to specify how many older versions of a file should be maintained). Labeled versions of a file are not deleted unless you delete them yourself, either from the File History dialog box or by deleting the file from the Advanced dialog box.

If you are using a third-party SCC system, you probably have the option of labeling files as well, but you might have to do it using the SCC tools that they provide. Also, notice that with third-party tools the way in which previous versions are maintained and when they get deleted is entirely configured using the tools that are provided with the third-party SCC tools.

If you are using the built-in SCC system and you need to label multiple files, you can use the File History dialog box on each one, but that can be a bit tedious. Instead, if you want to take a snapshot of all of the files in a project, applying the same label to all of the files, use the System History dialog box. It has an option that lets you label all files in a project with the same label. It also has an option for retrieving files with the same label.

Creating Reports

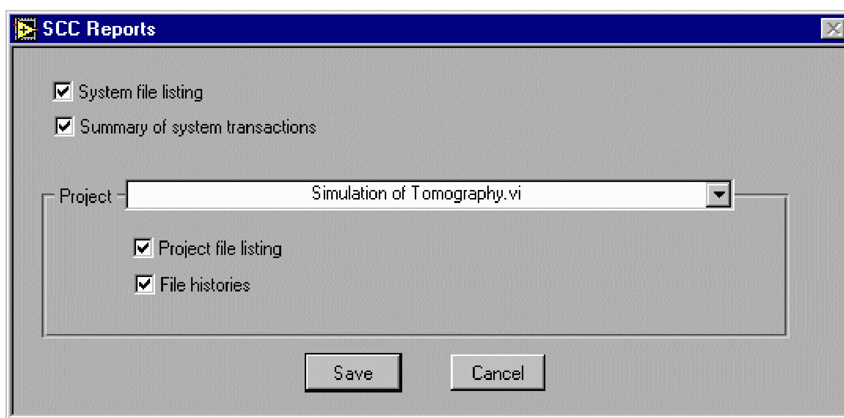
A very important feature of any SCC tool is the ability to generate reports describing system and file activity. This is because SCC tools should help you not only to maintain files but also to track the changes that happen to those files.

The **Reports** button in the **Project»Source Code Control»Advanced** dialog box lets you create basic reports describing file transactions and information about the projects maintained under Source Code Control. These reports are saved to a text file that you can edit or print using a word processor.

The options in the Reports dialog box depend upon the SCC system you are using (built-in or third-party).

Built-In System

If you are using the built-in system, you get the following dialog box.



The **System file listing** option describes all files maintained in Source Code Control. For each file, it lists the current file version and the projects that those files belong to.

The **Summary of system transactions** option gives the same information as in the System History dialog box. It lists all checkouts and checkins, project creations, file creations and file deletions. For each transaction, it includes the user name, the date, and a brief summary of the changes.

The **Project file listing** option lists only the files that are members of the selected project.

The **File histories** option lists the same information as in the File History dialog box for each file in the selected project. It lists information about when the file was first added to Source Code Control,

plus all information about subsequent changes to the file including the user name, the date, and a brief summary of the changes.

Microsoft Visual SourceSafe

With Visual SourceSafe, the file listings can be generated from the Reports dialog box. Transaction listings and other reports have to be done from the Visual SourceSafe environment. Remember that the system history can be viewed by looking at the history of the `sccfiles.lst` file (see the description of this file in the *System History* section).

Multiplatform Issues

Cross Platform Source Code Control

Unlike most Source Code Control tools, the built-in SCC system lets you access files from all the platforms that LabVIEW and BridgeVIEW support except Windows 3.1 (see the Using Individual Files Instead of LLBs topic for an explanation).

Once the administrator has set up the Source Code Control system, it can be accessed from any supported platform. There are some issues of which you should be aware if you are developing VIs on or for multiple platforms.

Filename Limitations

Macintosh, Windows 95 and NT, and Unix systems each have restrictions about filenames and paths that you need to be cautious about if you plan to support multiplatform development.

- Macintosh filenames are limited to 31 characters in length and cannot contain the ':' character. Paths are not limited in length.
- Unix filenames are limited to 255 characters in length and cannot contain the '/' character. Paths are not limited in length.
- Windows 95 filenames are limited to 255 characters in length and cannot contain any of the following characters: \, :, /, *, ?, ", <, >, and |. Paths are limited to 255 characters (including the filename). Windows NT supports both FAT and NTFS file systems. FAT filename restrictions are the same as Windows 95. NTFS filenames are limited to 255 characters in length and has the same character limitations as FAT. NTFS paths are not limited in length.

Consequently, for maximum portability you should avoid using :, \, /, *, ?, ", <, >, and | in filenames and limit filenames to 31 characters or less. Also, because of the path length restrictions under the FAT file system of Windows 95 and NT, you should avoid paths that are deeply nested (longer than 255 characters). The File Manager tool, described in Chapter 10, *File Manager Tool*, can scan a set of directories or VI libraries for invalid names.

Platform-Dependent SCC Files

LabVIEW and BridgeVIEW are available for a wide variety of computing platforms. Windows 95 and Power Macintosh are two examples of platforms for which LabVIEW is available. Most VIs can be opened on any platform LabVIEW and BridgeVIEW support and will run without modification. The G Source Code Control Tools can be used to share code among developers on any platform where LabVIEW or BridgeVIEW is available.

In most cases, you will probably want all files in Source Code Control to be available for all platforms. In some cases you may have files in your system that are platform-specific.

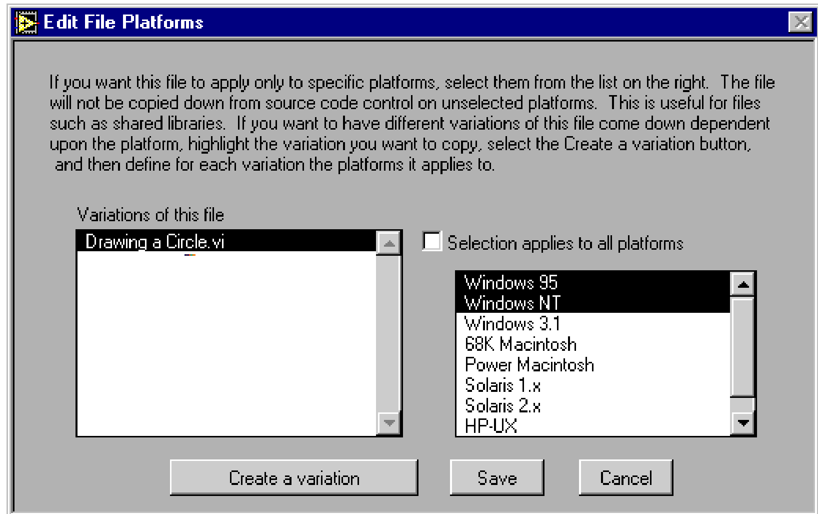
The following are cases that might involve platform-specific issues.

- VIs that take advantage of platform-specific features such as DDE can be taken to a platform that does not support the feature, but the VI will be broken on that platform. In this case, you might prefer to only have the file come down on the platforms that support the feature.
- If you write any VIs that contain CINs, you need a different version of the VI for each platform because CINs contain code compiled with platform-specific compilers.
- If your application uses shared libraries (DLLs), the libraries apply only to specific platforms. The VIs that call the libraries are platform independent assuming that you have a corresponding library for each platform

The G Source Code Control tools give you the flexibility of marking files as platform-specific and creating variants for different platforms.

Platform-Specific Files

From the **Source Code Control»Project** dialog box, select a project that contains the file you want to modify and press **Edit Project**. Then, from the Edit Project File List dialog box, select the **Platforms...** button to access a dialog box that lets you mark the platforms for which a file is available.



The platforms selected in the right list determine the platforms that the file is available on.

Each user of the G SCC tools will have configured the platform for which they want files using the **Source Code Control»Local Configuration** dialog box. Assuming you do not modify this list, the Configuration dialog box should have automatically detected your platform correctly.

Variants of a File for Different Platforms

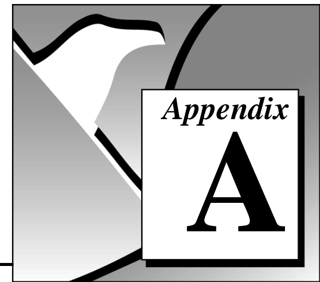
To create a variant of a file for another platform, go to the **Source Code Control»Project** dialog box, select a project that contains the file, press the **Edit Project** button, and press the **Platforms...** button. To create a new variant, press the **Create a Variation...** button. Then choose the new variant and select the platforms to which it applies. If necessary, modify the existing variants to ensure that there is no overlap (the dialog box does not allow you to apply multiple variants to the same platform).

Retrieving Files for a Different Platform

In some cases, you may need to retrieve files for a different platform. For example, if you create a CIN you will have different versions for each platform. Suppose you have a VI that calls a CIN, and at some point you decide to add a parameter to the CIN. You will first check out the VI on a platform, modify the CIN code, recompile and reload the CIN, and then check the VI back in. You might attempt to make the same modifications to the variant of the VI for each platform, but this would require a lot of work and is error prone. Instead, a better method is to check out the VI on each platform, replace it with the VI with the correct block diagram, and then recompile and then reload the CIN. Afterwards, you can check the resulting VI in as the variant for the current platform.

The Checkin, Checkout, and Retrieve Files dialog boxes restrict the list to the current platform. To change the platform, select **Source Code Control»Local Configuration** and then choose the platform for which you wish to retrieve files. After you have retrieved the files, remember to go back and reset the platform to its original value.

References



This appendix provides a list of references for further information about software engineering concepts.

LabVIEW Data Acquisition VI Reference Manual. A useful sample of quality documentation for libraries of VIs.

LabVIEW Instrument I/O Reference Manual. Detailed information for developers of instrument drivers.

Dataflow Programming with LabVIEW. National Instruments Application Note. A set of perspectives on dataflow programming that shows how LabVIEW compares with classical dataflow graphs, equations, and block diagrams.

Visual Programming Using Structured Data Flow. Jeffrey Kodosky, J. MacCrisken, G. Rymar, Proc. IEEE Workshop on Visual Languages, 1991. (Also available from National Instruments.) Description of some of the theory behind the graphical programming paradigm of LabVIEW.

LabVIEW Graphical Programming—Practical Applications in Instrumentation and Control. Gary W. Johnson, McGraw-Hill Inc., 1994, ISBN 0-07-032719-4. An excellent overview of how to apply LabVIEW to real-world problems.

LabVIEW Technical Resource. Editor: Lynda P. Gruggett, LTR Publishing, phone (214) 827-9931, fax (214) 827-9932. A quarterly newsletter and disk of VIs featuring technical articles about all aspects of LabVIEW.

Rapid Development. Steve McConnell, Microsoft Press. Explanation of software engineering practices in a very down-to-earth fashion with many examples and practical suggestions.

Microsoft Secrets. Michael A. Cusumano and Richard W. Selby, Free Press, 1995, ISBN 0-02-874048-3. In-depth examination of the programming practices Microsoft uses. Whether or not you are

interested in Microsoft, this book contains interesting discussions of what they have done right and what they have done wrong. Has a good discussion of team organization, scheduling, and milestones.

Dynamics of Software Development. Jim McCarthy, Microsoft Press, 1995, ISBN 1-55615-823-8. Another look at what has worked and what has not for developers at Microsoft. This book is written by a developer from Microsoft and contains numerous real-world stories that help put problems and solutions in focus.

Software Engineering—A Practitioner's Approach. Roger S. Pressman, McGraw-Hill, 1992, ISBN 0-07-050814-3. A detailed survey of software engineering techniques with descriptions of estimation techniques, testing approaches, and quality control techniques.

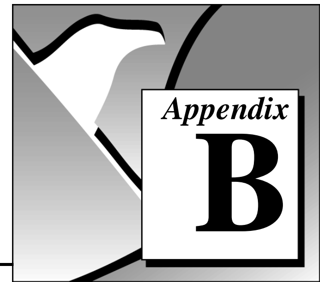
Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products. Daniel P. Freedman and Gerald M. Weinberg, Dorset House Publishing Co., Inc., 1990, ISBN: 9-932633-19-6. An excellent, down to earth discussion of how to conduct design and code reviews with many examples of things to look for and the best practices to follow during a review.

ISO 9000.3: A Tool for Software Product and Process Improvement. Raymond Kehoe and Alka Jarvis, Springer-Verlag New York, Inc., 1996, ISBN 0-387-94568-7. Describes what is expected by ISO 9001 in conjunction with ISO 9000.3 and provides templates for documentation.

Software Engineering Economics. Barry W. Boehm, Prentice Hall, 1981, ISBN 0-13-822122-7. Description of the delphi and COCOMO estimation techniques.

Software Engineering. Edited by Merlin Dorfman and Richard Thayer, IEEE Computer Science Press, 1996, ISBN 0-8186-7609-4. Collection of articles on a variety of software engineering topics, including a discussion of the spiral lifecycle model by Barry W. Boehm.

Customer Communication



For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at (512) 418-1111.

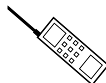
E-Mail Support (currently U.S. only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

	 Telephone	Fax
Australia	02 9874 4100	02 9874 4455
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 527 2321	09 502 2930
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 5734815	03 5734816
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ____

Company ____

Address ____

____

Fax (___)___________________ Phone (___) ________________________________

Computer brand ________________ Model ________________ Processor __________________

Operating system (include version number) ____________________________________

Clock speed ______MHz RAM _____MB Display adapter ___________________________

Mouse ___yes ___no Other adapters installed _________________________________

Hard disk capacity _____MB Brand ____

Instruments used ____

____

National Instruments hardware product model __________ Revision ______________________

Configuration ____

National Instruments software product ___________________________ Version ____________

Configuration ____

The problem is: ____

____

____

____

____

List any error messages: ____

____

____

The following steps reproduce the problem:____________________________________

____

____

____

____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *Professional G Developers Toolkit Reference Manual*

Edition Date: January 1997

Part Number: 321393A-01

Please comment on the completeness, clarity, and organization of the manual.

____
____
____
____
____
____
____

If you find errors in the manual, please record the page numbers and describe the errors.

____
____
____
____
____
____
____

Thank you for your help.

Name ____

Title ____

Company ____

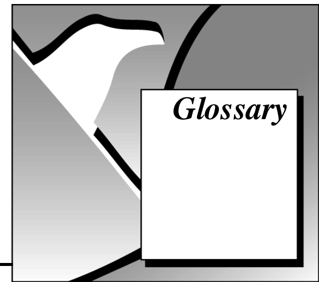
Address ____

____

Phone (___)____________________________ Fax (___)____________________________

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678



Prefix	Meaning	Value
m-	milli-	10^{-3}
μ-	micro-	10^{-6}
n-	nano-	10^{-9}

A

API — Application Programming Interface. The programming interface for controlling some software packages, such as Microsoft Visual SourceSafe.

B

black box testing — A form of testing where a module is tested without knowing how the module is implemented. The module is treated as if it were a black box that you cannot look inside. Instead, you generate tests that verify that the module behaves the way that it is supposed to according to the requirements specification.

C

CIN — *See* Code Interface Node.

Capability Maturity Model (CMM) — A model for judging the maturity of the processes of an organization and for identifying the key practices that are required to increase the maturity of these processes. The Software CMM (SW-CMM) has become a de facto standard for assessing and improving software processes. Through the SW-CMM, the Software Engineering Institute and software development community have put in place an effective means for modeling, defining, and measuring the maturity of the processes used by software professionals.

COCOMO estimation	COnstructive COst MOdel. A formula-based estimation method for converting software size estimates to estimated development time.
code and fix model	A lifecycle model that involves developing code with little or no planning, fixing problems as they arise.
Code Interface Node	A function in G that allows it to call compiled subroutines from other languages, such as C.
configuration management	A mechanism for controlling changes to source code, documents, and other material that make up a product. During software development, Source Code Control is a form of configuration management—changes only occur through the Source Code Control mechanism. It is also common to implement release configuration management, to ensure that a particular release of software can be rebuilt, if necessary. This implies archival development of tools, source code, and so on.

F

Function-Point Estimation	A formula-based estimation method applied to a category breakdown of project requirements.

I

integration testing	Integration testing assures that individual components work together correctly. Such testing may uncover, for example, a misunderstanding of the interface between modules.

L

lifecycle model	A model for software development, including steps to follow from the initial concept through the release, maintenance, and upgrading of the software.
LLB	LabVIEW VI Library.
LOC	Lines of Code. *See* Source Lines of Code.

P

prototype
: A simple, quick implementation of a particular task. It is used to demonstrate that the design has the potential to work. The prototype usually has missing features and might have design flaws. In general, prototypes should be thrown away, and the feature should be reimplemented for the final version.

R

race condition
: Race conditions occur when one block diagram reads from and writes to a global variable, and there is the potential that a parallel block diagram attempts to manipulate the same global variable, resulting in loss of data.

S

Source Lines of Code (SLOC)
: The measure of the number of lines of code that make up a project. It is used in some organizations to measure the complexity and cost of a project. How the lines are counted is organization-dependent. Some organizations do not count blank lines and comment lines, for example. Some count C lines; some count only the final assembly language lines.

SEI
: Software Engineering Institute, a federally funded research and development center chartered to address software engineering technology. The SEI is located at Carnegie Mellon University, and is sponsored by the Defense Advanced Research Projects Agency. See `http://www.sei.cmu.edu`.

spiral model
: A lifecycle model that emphasizes risk management through a series of iterations in which risks are identified, evaluated, and addressed.

system testing
: After integration testing is complete, system testing begins. System testing assures that all the individual components not only function correctly together, but that they comprise a product that meets the intended requirements. Performance, resource usage, and other problems are often uncovered at this stage.

U

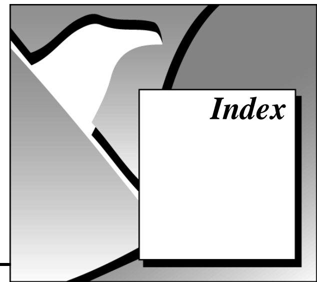
unit testing — Testing only a single component of a system, in isolation from the rest of the system. Unit testing occurs before the module is incorporated into the rest of the system.

W

waterfall model — A lifecycle model that consists of several non-overlapping stages, beginning with the software concept and continuing through testing and maintenance.

white box testing — Unlike black box testing, white box testing creates tests that take into account the particular implementation of the module. White box testing is used, for example, to verify that all the paths of execution of the module have been exercised.

wideband delphi estimation — Wideband delphi is a group estimation of techniques used to estimate the amount of effort a particular project will take.



Index

A

Administration. *See* Source Code Control tools.
alpha testing, 3-9 to 3-10
attribute nodes, 7-11

B

beta testing, 3-9 to 3-10
bibliography, A-1 to A-2
black box testing, 3-6
block diagram
statistics, 8-3 to 8-4
style considerations, 7-17 to 7-25
adding common threads, 7-19 to 7-20
checklist, 7-29
Code Interface Nodes (CINs), 7-25
data dependency, 7-19
error checking, 7-22 to 7-23
execution sequence, 7-19 to 7-21
labeling, 7-18 to 7-19
left-to-right layouts, 7-19
missing dependencies, 7-21
optimization, 7-24
sequence structures, 7-20 to 7-21
sizing and positioning, 7-23 to 7-24
wiring etiquette, 7-17 to 7-18
top-down design, 4-2 to 4-3
bottom-up design, 4-6 to 4-8
BridgeVIEW software, 1-1
bulletin board support, B-1

C

Capability Maturity Model (CMM) standards, 3-15 to 3-16
changes. *See* configuration management; Source Code Control tools.
CINs
CIN/shared library statistics, 8-4
description contents, 7-25
source code, 7-25
CMM (Capability Maturity Model) standards, 3-15 to 3-16
COCOMO (Constructive Cost Model) estimation, 5-6
code and fix model, 2-5
Code Interface Nodes (CINs). *See* CINs.
code walkthroughs, 3-12. *See also* design reviews.
coercion of invalid values, 7-10
color, style guidelines for, 7-5
common input/output terminal pairs, 7-19 to 7-20
common operations, identifying, 4-11 to 4-12
configuration management, 3-2 to 3-5. *See also* Source Code Control tools.
change control, 3-4 to 3-5
definition, 3-2
managing project-related files, 3-3
retrieving old versions of files, 3-3 to 3-4
source code control, 3-2 to 3-3
tracking changes, 3-4
connector panes, style considerations, 7-14 to 7-15
consistency of style. *See* style guidelines.
Constructive Cost Model (COCOMO) estimation, 5-6
control descriptions, as documentation, 6-6

controls and indicators
default values, ranges, and coercion, 7-10 to 7-11
indicator descriptions, as documentation, 6-6
local variables for consistent values, 7-12
style considerations, 7-8 to 7-12
attribute nodes, 7-11
default values, ranges, and coercion, 7-10 to 7-11
descriptions, 7-8
enumerations vs. rings, 7-9 to 7-10
key navigation, 7-11 to 7-12
labels, 7-8 to 7-9
local variables, 7-12
text styles, 7-5
cross-platform considerations. *See* multiplatform considerations.
custom controls and graphics, 7-6 to 7-7
customer communication, *xvi,* B-1 to B-2

D

data dependency, 7-19
missing dependencies, 7-21
default values for controls, 7-10 to 7-11
design-related documentation, 6-1 to 6-2
design reviews, 3-11. *See also* code walkthroughs.
design techniques, 4-1 to 4-9. *See also* development models.
bottom-up design, 4-6 to 4-8
data acquisition system (example), 4-4 to 4-6
defining requirements for application, 4-1 to 4-2
front panel prototyping, 4-10
identifying common operations, 4-11 to 4-12
instrument driver (example), 4-7 to 4-8
multiple developer considerations, 4-9
performance benchmarking, 4-11
top-down design, 4-2 to 4-6
development models, 2-1 to 2-12. *See also* design techniques; prototyping.
common pitfalls, 2-1 to 2-4
lifecycle models, 2-4 to 2-12
code and fix model, 2-5
definition, 2-4
G prototyping methods, 2-9
modified waterfall model, 2-8
prototyping, 2-8 to 2-9
spiral model, 2-10 to 2-12
waterfall model, 2-5 to 2-8
directories
converting LLBs to directories, 10-1 to 10-2
naming, 7-2
style considerations, 7-2
documentation for *Professional G Developer's Toolkit*
conventions used in manual, *xv*
organization of manual, *xiii-xiv*
references, A-1 to A-2
related documentation, *xvi*
documentation of applications, 6-1 to 6-6
BridgeVIEW and LabVIEW features, 6-2
design-related documentation, 6-1 to 6-2
help files, 6-4 to 6-5
overview, 6-1
Print Hierarchy tool, 9-1 to 9-2
user documentation, 6-2 to 6-4
application documentation, 6-3 to 6-4
library of subVIs, 6-2 to 6-3
VI and control descriptions, 6-5 to 6-6
control and indicator descriptions, 6-6
self-documenting front panels, 6-5 to 6-6
VI description, 6-5

E

e-mail support, B-2
Edit Platform List, 11-8 to 11-9

effort estimation, 5-4 to 5-6
COCOMO estimation, 5-6
function point estimation, 5-5 to 5-6
wideband Delphi estimation, 5-4 to 5-5
electronic support services, B-1 to B-2
enumerations vs. rings, 7-9 to 7-10
error checking, 7-22 to 7-23
estimation, 5-1 to 5-6
COCOMO estimation, 5-6
feature creep, 5-1
function point estimation, 5-5 to 5-6
lines of code/number of nodes, 5-2 to 5-3
mapping estimates to schedules, 5-6 to 5-7
of effort, 5-4 to 5-6
overview, 5-1 to 5-2
problems with size-based metrics, 5-3 to 5-4
wideband Delphi estimation, 5-4 to 5-5
execution sequence, 7-19 to 7-21
adding common threads, 7-19 to 7-20
data dependency, 7-19
left-to-right layouts, 7-19
missing dependencies, 7-21
sequence structures, 7-20 to 7-21

F

Fax-on-Demand support, B-2
FDA (U.S. Food & Drug Administration) standards, 3-14
feature creep, 5-1
file management. *See* Source Code Control tools.
File Manager tool, 10-1 to 10-2
filename limitations, on various platforms, 11-29 to 11-30
fonts, style guidelines, 7-5
Food & Drug Administration (FDA) standards, 3-14
front panels
self-documenting, 6-5 to 6-6
style checklist, 7-27 to 7-28
style considerations, 7-4 to 7-8
color, 7-6
consistency, 7-4
graphics and custom controls, 7-6 to 7-8
layout, 7-7
sizing and positioning, 7-8
text, 7-5
FTP support, B-1
function point estimation, 5-5 to 5-6

G

G Developers Toolkit. *See* Professional G Developers Toolkit.
G style guide. *See* style guidelines.
global/local statistics, 8-4
graphics and custom controls, 7-6 to 7-7

H

help files, creating, 6-4 to 6-5
hierarchical organization of files, 7-1 to 7-4
directories (folders), 7-1 to 7-2
naming VIs, VI libraries, and directories, 7-1 to 7-2
VI libraries, 7-3 to 7-4
History window, 6-2, 11-20
HP-UX installation, 1-3

I

icons, style considerations, 7-15 to 7-16
IEEE (Institute of Electrical and Electronic Engineers) standards, 3-16 to 3-17
indicators. *See* controls and indicators.
installation, 1-2 to 1-3
HP-UX, 1-3
Macintosh and Power Macintosh, 1-2
SPARCstation, 1-3
Windows 95 and NT, 1-2
Institute of Electrical and Electronic Engineers (IEEE) standards, 3-16 to 3-17

integration testing, 3-8 to 3-9
International Organization for Standards (ISO) 9000, 3-13 to 3-14

K

keyboard navigation, 7-11 to 7-12

L

labels
- block diagrams, 7-18 to 7-19
- font usage, 7-5
- style guidelines for, 7-8 to 7-9

LabVIEW software version required, 1-1
left-to-right layouts, 7-19
libraries. *See* VI libraries.
lifecycle models, 2-4 to 2-12
- code and fix model, 2-5
- definition, 2-4
- G prototyping methods, 2-9
- modified waterfall model, 2-8
- prototyping, 2-8 to 2-9
- spiral model, 2-10 to 2-12
- waterfall model, 2-5 to 2-8

lines of code. *See also* size-based metrics.
- in estimation, 5-2 to 5-3

LLBs. *See* VI libraries.
local configuration. *See* Source Code Control tools.
local variables
- global/local statistics, 8-4
- using for consistent values, 7-12

M

Macintosh and Power Macintosh installation, 1-2
manual. *See* documentation for *Professional G Developer's Toolkit.*
master file list (sccfiles.lst), 11-25 to 11-26
metrics. *See* size-based metrics; VI Metrics tool.
Microsoft Visual SourceSafe for Windows 95/NT
- accessing previous versions of files, 11-27
- administration, 11-6
- overview, 11-5
- report generation, 11-29
- user configuration, 11-9

milestones
- responding to missed milestones, 5-8
- tracking schedules using milestones, 5-7 to 5-8

missing dependencies, 7-21
modified waterfall model, 2-8
multiplatform considerations, 11-29 to 11-32
- cross platform source code control, 11-29
- Edit Platform List, 11-8 to 11-9
- filename limitations, 11-29 to 11-30
- platform-dependent SCC files, 11-30
- platform-specific files, 11-31
- retrieving files for different platforms, 11-32
- variants of files for different platforms, 11-31
- work directory and platform configuration, 11-10

N

naming VIs, VI libraries, and directories, 7-1 to 7-2
nodes. *See also* size-based metrics.
- number of, 5-3, 8-2

O

optimizing programs, 7-24

P

performance benchmarking, 4-11
platforms. *See* multiplatform considerations.
postmortem evaluation, 3-13

Print documentation dialog box, 6-2
Print Hierarchy tool, 6-2, 9-1 to 9-2
Professional G Developers Toolkit
  features, 1-5
  installation, 1-2 to 1-3
  overview, 1-4
  required system configuration, 1-1
project management. *See* scheduling and project tracking; Source Code Control tools.
prototyping. *See also* design techniques.
  development model, 2-8 to 2-9
  front panel prototyping, 4-10
  G prototyping methods, 2-9

Q

quality control, 3-1 to 3-17
  code walkthroughs, 3-12
  configuration management, 3-2 to 3-5
    change control, 3-4 to 3-5
    managing project-related files, 3-3
    retrieving old versions of files, 3-3 to 3-4
    source code control, 3-2 to 3-3
    tracking changes, 3-4
  design reviews, 3-11
  postmortem evaluation, 3-13
  requirements, 3-1 to 3-2
  software quality standards, 3-13 to 3-17
    CMM, 3-15 to 3-16
    FDA standards, 3-14
    IEEE, 3-16 to 3-17
    ISO 9000, 3-13 to 3-14
  style guidelines, 3-10 to 3-11
  testing guidelines, 3-5 to 3-10
    black box and white box testing, 3-6
    formal methods of verification, 3-10
    integration testing, 3-8 to 3-9
    system testing, 3-9 to 3-10
    unit testing, 3-7 to 3-8

R

ranges of values for controls, 7-10 to 7-11
references, A-1 to A-2
report generation with SCC tools, 6-2, 11-27
required system configuration, 1-1
rings vs. enumerations, 7-9 to 7-10
risk management. *See* spiral model.

S

safeguarding applications, 3-1 to 3-2. *See also* quality control.
SCC. *See* Source Code Control tools.
scheduling and project tracking, 5-1 to 5-8. *See also* Source Code Control tools; VI Metrics tool.
  estimation, 5-1 to 5-6
    COCOMO estimation, 5-6
    function point estimation, 5-5 to 5-6
    lines of code/number of nodes, 5-2 to 5-3
    of effort, 5-4 to 5-6
    problems with size-based metrics, 5-3 to 5-4
    wideband Delphi estimation, 5-4 to 5-5
  mapping estimates to schedules, 5-6 to 5-7
  tracking schedules using milestones, 5-7 to 5-8
    missed milestones, 5-8
sequence structures, 7-20 to 7-21
size-based metrics. *See also* VI Metrics tool.
  lines of codes, 5-2 to 5-3
  number of nodes, 5-3
  problems, 5-3 to 5-4
software quality standards, 3-13 to 3-17
  CMM, 3-15 to 3-16
  FDA standards, 3-14
  IEEE, 3-16 to 3-17
  ISO 9000, 3-13 to 3-14

Source Code Control tools, 11-1 to 11-32
  administration, 11-5 to 11-9
    built-in system, 11-7
    Edit Platform List, 11-8 to 11-9
    Visual SourceSafe, 11-6
  advanced features, 11-22 to 11-29
  configuration, 11-2 to 11-10
    administration, 11-5 to 11-9
    local configuration (all users), 11-9 to 11-10
    selecting system for source code control, 11-3 to 11-5
    work directory and platform configuration, 11-10
  features
    built-in Source Code Control System, 11-3 to 11-4
    Microsoft Visual SourceSafe for Windows 95/NT, 11-5
    third-party systems, 11-4 to 11-5
  file management, 11-16 to 11-22
    accessing previous versions of files, 3-3 to 3-4, 11-26 to 11-27
    change control, 3-4 to 3-5
    checking in files, 11-20 to 11-22
    checking out files, 11-18 to 11-20
    deleting files from SCC, 11-23
    file properties, 11-18
    file status, 11-17
    History window for documenting changes, 11-20
    labeling versions of files for easy retrieval, 11-27
    managing project-related files, 3-3
    master file list (sccfiles.lst), 11-25 to 11-26
    retrieving files, 11-16 to 11-18
    SCC user name, 11-22
    tracking changes, 3-4
  general concepts, 11-1
  multiplatform issues, 11-29 to 11-32
    cross platform source code control, 11-29
    filename limitations, 11-29 to 11-30
    platform-dependent SCC files, 11-30
    platform-specific files, 11-31
    retrieving files for different platforms, 11-32
    variants of files for different platforms, 11-31
  overview, 3-3
  project management, 11-10 to 11-16
    adding extra files, 11-14 to 11-15
    creating projects, 11-11 to 11-13
    overview, 11-10 to 11-11
    project groups, 11-15 to 11-16
    removing files from projects, 11-13 to 11-14
    updating projects, 11-13
  quality control considerations, 3-2 to 3-3
  report generation, 11-27 to 11-29
  SCC File History, 11-24 to 11-25
  System History, 11-25 to 11-26
  user configuration
    built-in system, 11-9 to 11-10
    Visual SourceSafe, 11-9
  using files instead of VI libraries, 11-2
Source Lines of Code (SLOCs) metric, 8-1. *See also* size-based metrics.
SPARCstation installation, 1-3
spiral model, 2-10 to 2-12
standards. *See* software quality standards.
statistics. *See* VI Metrics tool.
stub VIs, 4-9
style guidelines, 7-1 to 7-29
  block diagram, 7-17 to 7-25
    adding common threads, 7-19 to 7-20
    Code Interface Nodes (CINs), 7-25
    data dependency, 7-19
    error checking, 7-22 to 7-23
    execution sequence, 7-19 to 7-21
    labeling, 7-18 to 7-19
    left-to-right layouts, 7-19
    missing dependencies, 7-21
    optimization, 7-24

sequence structures, 7-20 to 7-21
sizing and positioning, 7-23 to 7-24
wiring etiquette, 7-17 to 7-18
connector panes, 7-14 to 7-15
controls and indicators, 7-8 to 7-12
attribute nodes, 7-11
default values, ranges, and coercion, 7-10 to 7-11
descriptions, 7-8
enumerations vs. rings, 7-9 to 7-10
key navigation, 7-11 to 7-12
labels, 7-8 to 7-9
local variables, 7-12
front panels, 7-4 to 7-8
color, 7-6
consistency, 7-4
graphics and custom controls, 7-6 to 7-8
layout, 7-7
sizing and positioning, 7-8
text, 7-5
hierarchical organization of files, 7-1 to 7-4
directories (folders), 7-1 to 7-2
naming VIs, VI libraries, and directories, 7-1 to 7-2
VI libraries, 7-3 to 7-4
icons, 7-15 to 7-16
problems with inconsistent developer styles, 3-10 to 3-11
style checklist, 7-26 to 7-29
block diagram, 7-29
front panel, 7-27 to 7-28
VIs, 7-26 to 7-27
VI setup, 7-13
subVI interface statistics, 8-5
subVI library, documenting, 6-2 to 6-3
System History dialog box, 11-25
system testing, 3-9 to 3-10

T

technical support, B-1 to B-2
telephone and fax support, B-2
testing guidelines, 3-5 to 3-10
black box and white box testing, 3-6
formal methods of verification, 3-10
integration testing, 3-8 to 3-9
system testing, 3-9 to 3-10
unit testing, 3-7 to 3-8
text, style guidelines, 7-5
top-down design, 4-2 to 4-3
tracking changes, 3-4
tracking projects. *See* scheduling and project tracking; Source Code Control tools.

U

unit testing, 3-7 to 3-8
U.S. Food & Drug Administration (FDA) standards, 3-14
user documentation. *See* documentation of applications.
user interface statistics, 8-4

V

verification methods, 3-10. *See also* testing guidelines.
VI libraries
avoiding with Source Code Control tools, 11-2
converting LLBs to directories, 10-1 to 10-2
documenting subVI libraries, 6-2 to 6-3
File Manager tool, 10-1 to 10-2
files in vi.lib excluded from VI Metrics tool, 8-5
hierarchy with VI libraries, 7-3 to 7-4
Windows 3.1 considerations, 7-3
VI Metrics tool, 8-1 to 8-5
dialog box, 8-1
files in vi.lib, 8-5
number of nodes, 8-2

purpose and use, 8-1 to 8-2
saving metric information, 8-5
statistics, 8-3 to 8-5
block diagrams, 8-3 to 8-4
CIN/shared library statistics, 8-4
global/local statistics, 8-4
subVI interface statistics, 8-5
user interface, 8-4
VI setup, 7-13
VIs
description, as documentation, 6-5
hierarchy on disk, 7-1 to 7-2
style checklist, 7-26 to 7-27
Visual SourceSafe for Windows 95/NT. *See* Microsoft Visual SourceSafe for Windows 95/NT.

W

waterfall model, 2-5 to 2-8
modified, 2-8
white box testing, 3-6
wideband Delphi estimation, 5-4 to 5-5
Windows 3.1
restrictions on development (note), 1-1
saving VIs in libraries, 7-3
Source Code Control tools unavailable, 11-2
Windows 95 and NT installation, 1-2
wiring tips, 7-17 to 7-18