BridgeVIEW™ and LabVIEW™

Professional G Developers Tools Reference Manual

NATIONAL INSTRUMENTS™

May 1998 Edition
Part Number 321393B-01

Internet Support
E-mail: support@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 (USA)
Tel: 512 795 8248
Fax: 512 794 5678

International Offices
Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11,
France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, 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,
United Kingdom 01635 523545

National Instruments Corporate Headquarters
6504 Bridge Point Parkway Austin, Texas 78730-5039 USA 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

Trademarks

BridgeVIEW™, LabVIEW™, `natinst.com`™, and National Instruments™ 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.

Contents

Chapter 3
Incorporating Quality into the Development Process

Chapter 4
Prototyping and Design Techniques

Chapter 5
Scheduling and Project Tracking

Chapter 6
Creating Documentation

Chapter 7
Using Consistent Style: The G Style Guide

PART II
Professional Development Tools

Chapter 8
VI Metrics Tool

Chapter 9
Documentation Tool

Chapter 10
VI Comparison Tools

Chapter 11
Source Code Control Tools

Figures

Tables

About This Manual

The *Professional G Developers Tools Reference Manual* describes the features, functions, and operation of the Professional G Developers Tools. With these tools, you can apply software engineering techniques to G code development. This package adds important software engineering tools to LabVIEW and BridgeVIEW.

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 Tools Reference Manual* is divided into two sections. Chapters 2 through 7 describe software engineering concepts. Chapters 8 through 11 describe the tools.

- Chapter 1, *Introduction*, introduces you to the features of the Professional G Developers Tools.

Part I—Software Engineering Concepts

- 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 estimating 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 recommended practices for good programming technique and style.

Part II—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, *Documentation Tool*, describes to create documentation for VIs in HTML or Rich Text Format (RTF), to create source material for online help files, or to print the material directly to a printer.
- Chapter 10, *VI Comparison Tools*, describes the VI Comparison tools, which you can use to manage different versions of VIs as you develop large applications.
- Chapter 11, *Source Code Control Tools*, describes the G Source Code Control (SCC) tools, which allow you to add files to SCC and access those files from within the LabVIEW or BridgeVIEW environment.

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 and acronyms.
- 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:

<> Angle brackets enclose the name of a key on the keyboard—for example, <Shift>.

- A hyphen between two or more key names enclosed in angle brackets denotes you should simultaneously press the named keys—for example, <Control-Alt-Delete>.

» 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.
☞	This icon to the left of bold italic text denotes a note, which alerts you to important information.
bold	Bold text denotes the names of menus, menu items, parameters, dialog boxes, dialog box buttons or options, icons, palettes, or windows.
bold italic	Bold italic text denotes a note.
<Control>	Key names are capitalized.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text from which you supply the appropriate word or value, as in Windows 3.*x*.
`monospace`	Text in this font denotes text or characters you should literally enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and for statements and comments taken from programs.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.
Platform	Text in this font denotes information related to a specific platform.

Related Documentation

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

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

Refer to 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.

1

Introduction

This chapter introduces you to the features of the Professional G Developers Tools.

Required System Configuration

These tools are 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** ***Although you cannot use these tools under Windows 3.1, you still can develop for customers who need Windows 3.1 support. You can use Windows 95/NT as the development platform and save VIs as individual files. When you need to send software to a Windows 3.1 user, save VIs in LLBs.***

Configuration

For installation instructions, refer to the *Professional G Developers Tools Release Notes*. After installation is complete, the administrator must set up the Source Code Control system for the other users. All users must perform local configuration. Refer to the *Administrator Setup* and *Local Configuration* sections in Chapter 11, *Source Code Control Tools*, for more information.

Overview

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

National Instruments designed G as an easy-to-use, general-purpose language. Because G is a fully functional programming language, LabVIEW and BridgeVIEW deal with 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 that consists of a front panel that provides a user interface and a block diagram that represents the source code. The block diagram describes the relationship and interactions between inputs and outputs in the user interface. Every VI is a reusable component. You can define a calling interface and a representative icon so you can call the VI as a subroutine, or subVI, from other VIs.

Because VIs can be stored in separate files, multiple developers can 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 simple applications require. 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. These tools are designed to help users apply these techniques to G code development. Most software engineering techniques apply to graphical programming languages just as well as they apply to textual programming languages.

Features of the Tools

The Professional G Developers Tools are designed to simplify development of high-end, large-scale applications. The tools can help you manage and track 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 the U. S. Food and Drug Administration.

The tools include features that help you do the following:

- Control source code—Integrated Source Code Control 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* so they can communicate with either a built-in SCC system available to all platforms or other third-party SCC systems. These tools support the built-in system, Microsoft Visual SourceSafe for Windows 95/NT, and Rational Software ClearCase for Solaris 2.
- 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* metrics for textual languages. With this tool, you also can view many other statistics about VIs, which are useful in examining your VIs to find overly complex areas or in establishing baselines for estimating future projects.
- Print documentation—The Documentation tool makes it easy to print documentation for VIs in your hierarchy. In addition to creating printed documentation, you can use this tool to create Web pages, online help source files, and word-processor documents for your VIs.
- Compare files—VI Comparison tools make it easy to view the differences between two VIs, two hierarchies of VIs, and between files and files under Source Code Control. This is extremely important when multiple developers work on the same set of VIs or when you need to understand how your VIs have changed.

Part I

Software Engineering Concepts

This section of the manual describes software engineering concepts.

- 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 estimating 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 recommended practices for good programming technique and style.

2

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 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 statements. 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. Refer to chapter Chapter 5, *Scheduling and Project Tracking*, for more information on techniques for estimation.

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

  There are two problems with this statement. 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 in the *Lifecycle Models* section later in this chapter, 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. Refer to 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. Refer to 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 you analyze 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. Refer to the *Lifecycle Models* section later in this chapter for more information.

- "If I can just get all 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, you should 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. Although you might complete all the features on time, the time required to correct the problems in the existing and the 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. Refer to Chapter 3, *Incorporating Quality into the Development Process*, for more information 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 can 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. Assuming you can make up lost time can postpone 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 you do not realize you are behind schedule until the fifth month, other groups might have made decisions that are costly to change.

  When you realize you are behind, adjust the schedule or consider features you can drop or postpone to subsequent releases. Do not ignore the delay or sacrifice testing scheduled for later in the process. Refer to Chapter 5, *Scheduling and Project Tracking*, for more information on estimating project schedules.

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 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.

Although this section is theoretical in its discussion, in practice you should consider all the steps these models encompass. You should consider when and how you decide that the requirements and specifications are complete and how you deal with 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 easier and much less expensive to make changes.

The code and fix model is appropriate only 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, as shown in Figure 2-1. It begins with the software concept and continues through requirements analysis, architectural design, detailed design, coding, testing, and maintenance.

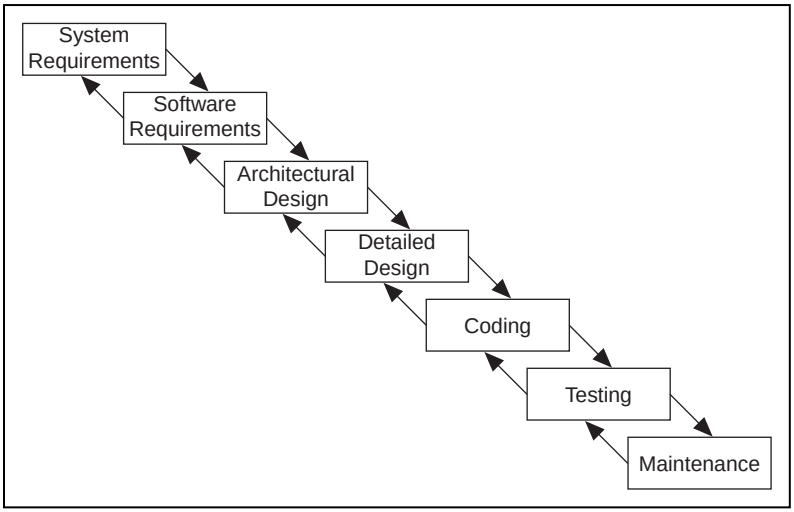


Figure 2-1. Waterfall Lifecycle Model

- 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 the software affects. 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. You also 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 deal with 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. Refer to 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 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 you begin the design, and you must state the complete design before you begin coding. 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 to

customers. Many people also think the amount of documentation is excessive and inflexible.

Although 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 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 following 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. 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 often are 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 you need to adjust requirements.

Prototyping can be an effective tool for demonstrating how a design might deal with a set of requirements. You can build a *prototype*, 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 allows for prototyping in the later architectural design stage and subsequent stages, but not in the early requirements stages.

Prototyping has drawbacks, however. Because it appears that you have a working system 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 that 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 to 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 returning 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. In I/O prototypes, random data can simulate data acquired in the real system.

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.

If you are bidding on a project for a client, using front panel prototypes 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 dealt with. You then begin a series of iterations in which you analyze the most important risk, evaluate options for resolving the risk, deal with the risk, assess the results, and plan for the next iteration. Figure 2-2 illustrates the spiral lifecycle model.

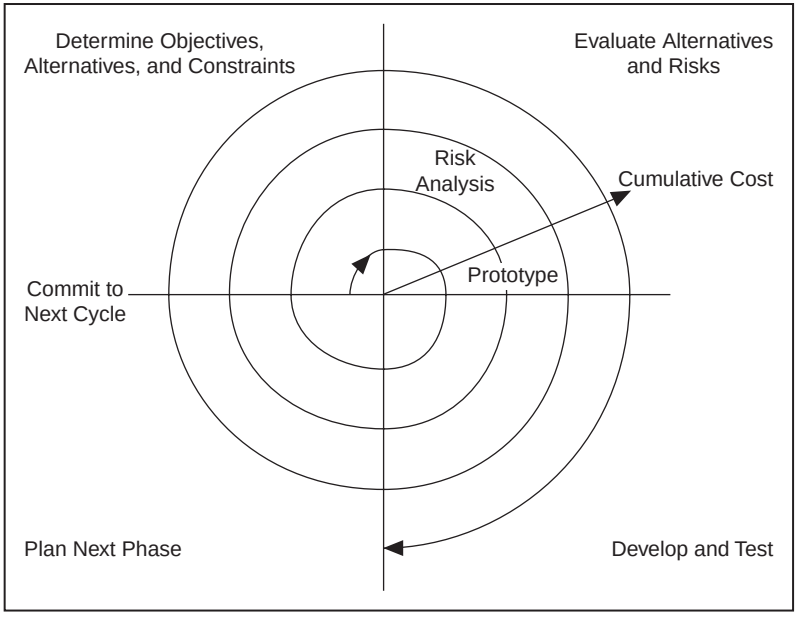


Figure 2-2. Spiral Lifecycle Model

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 to occur (probability) and the severity of its effect on the project (loss). You might use a scale of 1 to 10 for each of these items, with 1 representing the lowest and 10 representing the highest. Your risk exposure is the product of these two rankings.

You can use a table to keep track of the top risk items of your project. Table 2-1 gives 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 might 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 deal with the risks with the highest risk exposure first. In this example, the first spiral should deal with the potential of the data acquisition rates being too high. After the first spiral, you might 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 higher cost a new, more likely risk.

For example, assume 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 system cost and requirements for a specific sampling rate and precision.

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 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 take care of with each cycle. Because you can evaluate risks with prototypes much earlier than in the waterfall process, you can deal with 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, which requires 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 specifications 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.

Other lifecycle models exist. Appendix A, *References*, lists documents that contain information about other development methodologies.

3

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 they do not need to deal with the issue of quality 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 problems are found before they are in the software, when it is inexpensive to fix them.

Quality Requirements

Set the quality standards for your product during the requirements stage. 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 include speed versus robustness, and ease-of-use versus power and complexity.

For short projects that will be used only 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, error checking is not as crucial.

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

should deal with 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 they are reviewed before they are made. Chapter 2, *Development Models*, outlines development models, such as the waterfall model. 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 deal with changes. This is 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 deal with this is to establish a central source repository. If each of the developer's computers is networked, you can create a shared location that serves as a central source for development. When developers 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 easily merge changes into the project. The other developer will have to use the VI Comparison tool to determine the differences and merge the changes into a new version. 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 deal with 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 again. Another developer can then check out the file to make further modifications.

The G Source Code Control tools are accessible from the **Project** menu. Refer to Chapter 11, *Source Code Control Tools*, for more information on these 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. 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. In fact, source management is a requirement for certification under 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 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 the log for that file.

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

In addition, if you back up your project at specific checkpoints, you can use the VI Comparison tools to compare the latest version of a project with another version to verify the changes in your project. Refer to Chapter 10, *VI Comparison 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 you introduce into your system.

Changes to specific components, such as documents related to user requirements, must be dealt with cautiously because they generally are worked out through several iterations with the customer. In this case, the word *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 when making changes. You can trace all changes, and you can configure the system to maintain previous versions so 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 or Rational Software ClearCase for Solaris 2, you can control access to files so some users have access to specific files but others do not. You also can specify that anyone can retrieve files but only certain users can make modifications.

With this kind of access control, you might limit change privileges for requirement documents to specific team members. Or, you might control access so a user has privileges to modify 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.

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.

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 the system and all 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 so on.

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

The method of *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 deals with it well. For example, for numeric inputs, you should see how the subVI deals with infinity, not a number, and other out-of-range values. Refer to the *Unit Testing* section later in this chapter for more examples.

The method of *white box testing* is designed with knowledge of the internal workings of the software. Use white box testing to check that all 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.

Although 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

You can use *unit testing* to concentrate on testing individual software components. For example, you might test an individual VI to see that it works correctly, deals with 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 include the following:

- Boundary conditions for each input, such as empty arrays and empty strings, or 0 for a size input. Be sure floating point parameters deal with 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 pathnames.
- 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 and write a test VI that calls your VI with each combination of inputs and checks the results. You can use interactive data logging to create your input sets, or 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 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 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 following *Integration Testing* 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, and it is also useful 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

You perform *integration testing* on a combination of units. Unit testing usually finds most bugs, but 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, refer to Chapter 28, *Performance Issues*, in the *G Programming Reference 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 to 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 expensive because it is difficult to determine the specific source of problems within a large set of VIs. Instead, you should consider testing incrementally 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 earlier in this chapter. 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 in the *Unit Testing* section earlier in this chapter. 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 to use 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 the software works as expected within the hardware system. This can be done first as a set of black box tests to 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 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 deal with 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. Other testing methodologies attempt to find problems by exploration, but formal methods attempt to *prove* the correctness of software mathematically.

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. Appendix A, *References*, lists documents that contain information on this process.

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. Users might become confused and find the user interface VIs difficult to use if the VIs have different behaviors, such as some expecting a user to click a button when he or she is finished and others expecting the user to use a keyboard function key.

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, that VI 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. You can use these style guidelines 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. A single standard for programming style in any language really cannot exist because 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. There is nothing wrong with building in extra features. If quality and schedule are important, however, you should ensure that these extra features are scheduled for late in the development process, so they can be dropped, or moved to the list of features for subsequent releases. Document the results of the design review and any recommended changes.

Code Walkthroughs

A code walkthrough is similar to a design review except that it analyzes the code instead of the design. To perform a code review, give one or more developers printouts of the VIs to review. 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 a specific VI or function returns an error? Are errors dealt with 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. There is the potential a parallel block diagram could simultaneously attempt to manipulate the same global variable, resulting in loss of data.

- Is the block diagram implemented well? Are the algorithms efficient in terms of speed and/or memory usage? For more information, refer to Chapter 28, *Performance Issues*, in the *G Programming Reference 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? Does the developer adhere to established guidelines?

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.

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

Appendix A, *References*, lists documents that contain information on walkthrough techniques.

Post-Mortem Evaluation

At the end of each stage in the development process, you should consider having a post-mortem meeting to discuss what has gone well and what has not. Each developer should evaluate the project and be honest 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 we improve?
- 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?

Post-mortem 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 are specific to software or that 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 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. However, it is most commonly referred to as ISO 9000 in all countries.

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 standard 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 specifically 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 and Drug Administration Standards

The U.S. Food and Drug Administration (FDA) requires all software used in medical applications to meet its Current Good Manufacturing Practices (CGMP). One of the goals of the standard is to make it as consistent as possible with ISO 9001 and a supplement to ISO 9001, ISO/CD 13485. These FDA standards are largely consistent with ISO 9001, but there are some differences. Specifically, the FDA did not think ISO 9001 was specific enough about certain requirements, so the FDA clearly outlined them in its rules.

Refer to the FDA Internet home page at `http://www.fda.gov` for more information on the new CGMP rules and how they compare to ISO 9001.

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—The organization establishes 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. Although the groups can deal with similar projects based on this experience, their processes might not be mature enough to deal with 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.

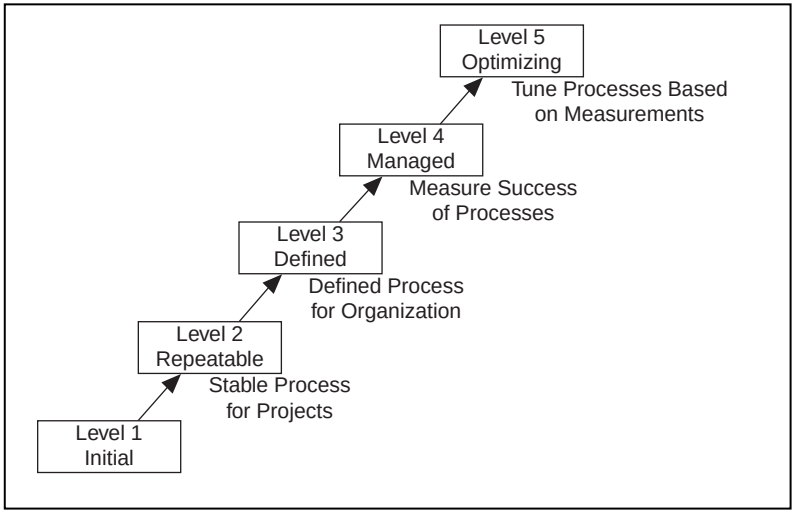


Figure 3-1. Capability Maturity Model

Most companies are at Level 1 or 2. The U.S. Department of Defense prefers a Level 3 or higher CMM assessment in bids on new government software development. Some commercial companies, mainly in the United States, also use the CMM.

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

Institute of Electrical and Electronic Engineers (IEEE) Standards

IEEE defined a number of standards for software engineering. IEEE Standard 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 730 is fairly short. It does not dictate how to meet the requirements but requires documentation for these practices to a specified minimum level of detail.

In addition to IEEE 730, several other IEEE standards related to software engineering exist, including the following:

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

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

4

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 some of the VIs they think they will need. Then they realize 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 you develop 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 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, when you design an I/O system, the customer probably has certain sampling rate and precision requirements. He or she also is constrained by cost. You should include those issues 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 difficulties 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 G uses 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 determine the high-level blocks, create a block diagram that uses those blocks. For each block, create a new *stub* VI, which is a non-functional prototype that represents 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 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. Refer to Chapter 28, *Performance Issues*, of the *G Programming Reference 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 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.

Data Acquisition System Example

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

Start to design the VI hierarchy by breaking the problem into logical pieces. The flowchart in Figure 4-1 shows several major blocks you can expect to see in one form or another in every data acquisition system.

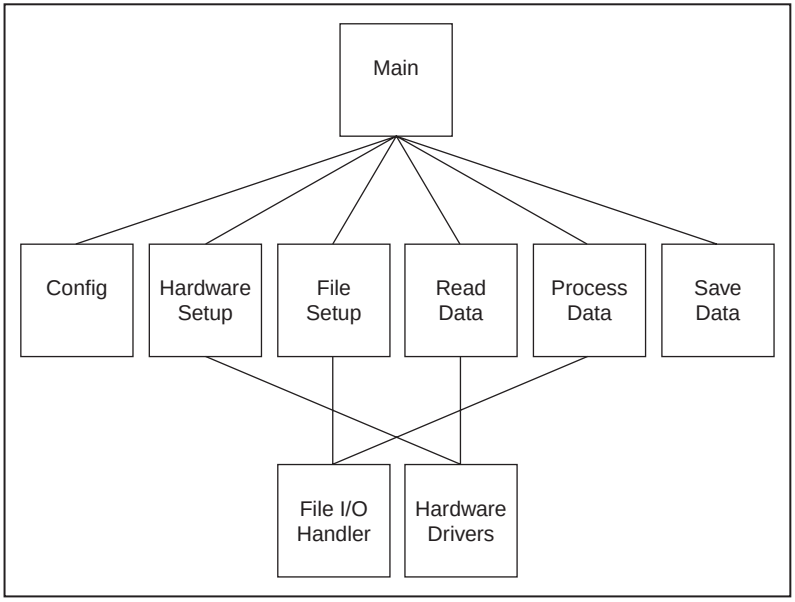


Figure 4-1. Flowchart of a Data Acquisition System

Think about the data structures 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, which is an array of many channels, each element of which is a cluster that contains the value, the channel name, scale factors, and so on. A method that performs some action on such a data structure is called an algorithm. Algorithms and data structures are intertwined. This is reflected in modern structured programming, and it works well in G. If you like to use pseudocode, try that technique as well. Figures 4-2 and 4-3 show a relationship between pseudocode and G structures.

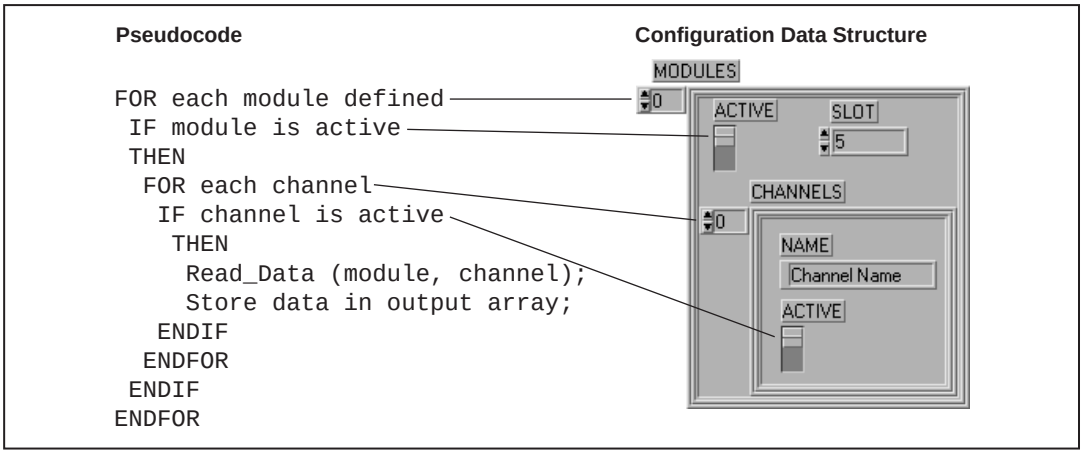


Figure 4-2. Mapping Pseudocode into a G Data Structure

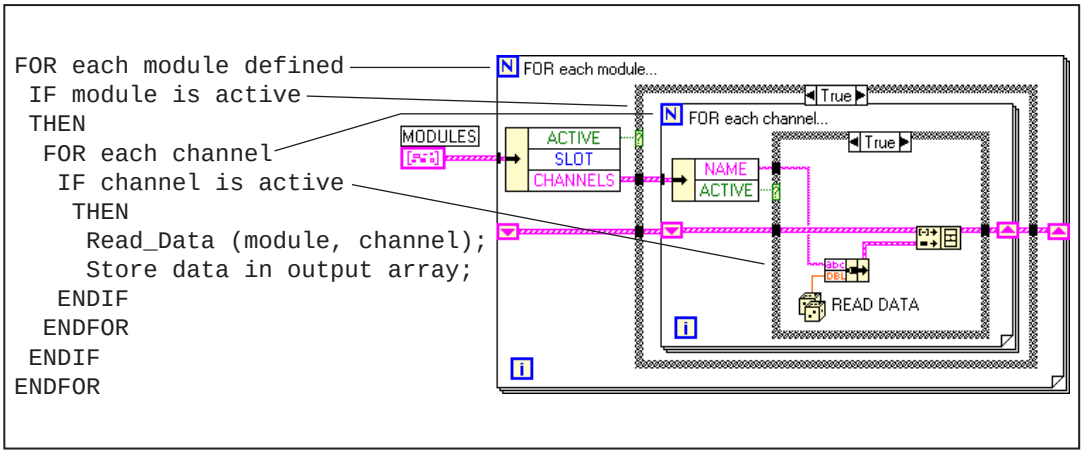


Figure 4-3. Mapping Pseudocode into Actual G Code

Notice that the program and the data structure correspond.

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, as shown in Figure 4-4. Artificial data dependency between the initialization VIs and the main While Structure in Figure 4-4 eliminates the need for a Sequence Structure.

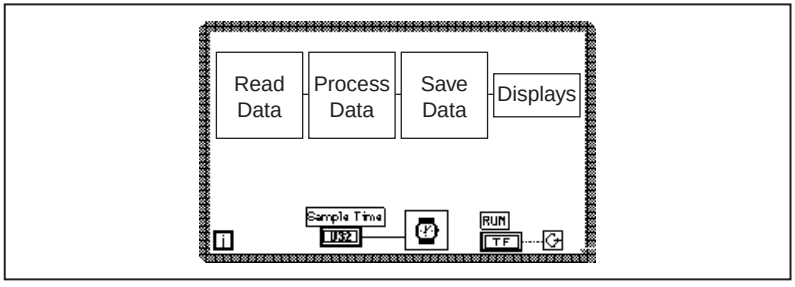


Figure 4-4. Data Flow for a Generic Data Acquisition Program

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 constructing 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. 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 progressing up the hierarchy, gradually putting pieces together until you have the complete system.

The problem with bottom-up design is that because you do not start with a clear idea of the big picture, 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 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 you could 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.

Instrument Driver Example

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 way to design 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 requires a complicated parser for the waveform preamble that contains information such as scale factors, offsets, sources, and units. It is 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 makes it 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.

Chapter 7, *Getting Started with a LabVIEW Instrument Driver*, of the *LabVIEW User Manual* describes this development process in detail.

The top-level VI in Figure 4-5 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.

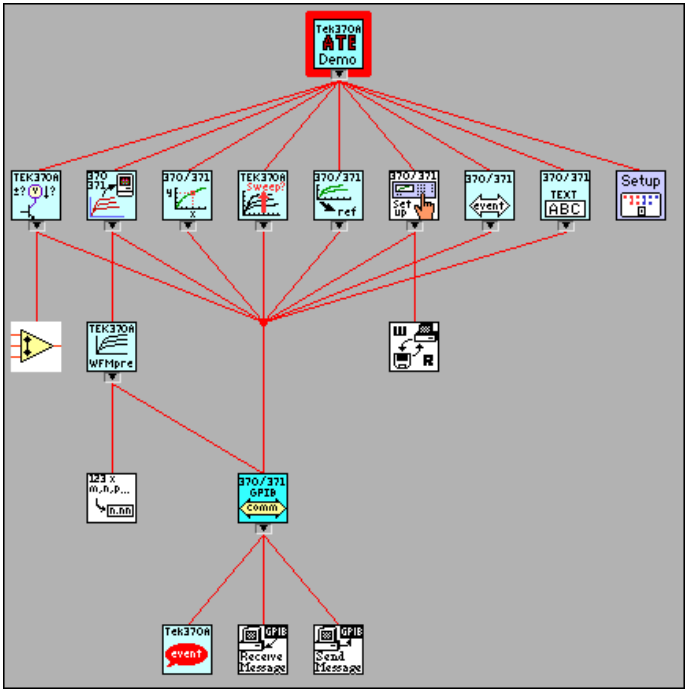


Figure 4-5. VI Hierarchy for the Tektronix 370A

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 to display and manipulate data and another set to acquire the information and transfer 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 to verify the system works as a whole. Refer to the *Integration Testing* section in Chapter 3, *Incorporating Quality into the Development Process*, for more information.

Front Panel Prototyping

As mentioned in 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 create a mock 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. As you create buttons, list boxes, and rings, think about what should happen as the user makes selections. Ask yourself 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 deals only 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 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 deal with all these issues. For those issues, you need to use one of the other methods described in this chapter, such as performance benchmarking and top-down design.

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, that switching between channels with different gains cannot be done at the necessary rate without reducing the accuracy of the sampling, or that 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 reveal this kind of problem. The timing template example in the `examples/general/structs.llb` directory illustrates how to time a process. Because timings can fluctuate from one run to another for a variety of reasons, you should put the operation in a loop and display the average execution time. You also can use a graph to display timing fluctuations. Causes of timing fluctuations can include system interrupts, screen updates, user interaction, and initial buffer allocation.

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 Figure 4-6, where three similar operations run independently.

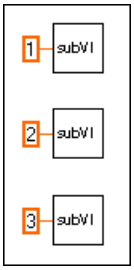


Figure 4-6. Operations Run Independently

An alternative to this design is a loop that performs the operation three times, as shown in Figure 4-7. You can build an array of the different arguments and use auto-indexing to set the correct value for each iteration of the loop.

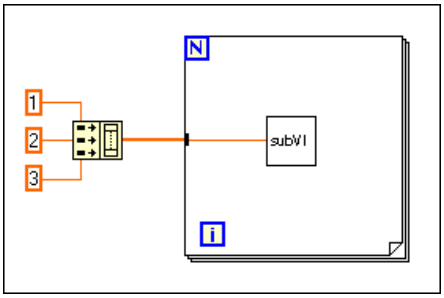


Figure 4-7. Loop Performs Operation Three Times

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.

5

Scheduling and Project Tracking

This chapter describes techniques for estimating 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, or 8 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, such as meetings, support for existing projects, and so on.

Estimation

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

Unfortunately, when it comes to estimating software schedules accurately, few people are successful. Major companies 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 *feature creep*: Your design gradually grows to include features that were not part of the original requirements. In many cases, the delays in schedule are a result of using 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 2 months at first might seem like an infinite amount of time. During the last 2 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 2 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 and conferences to attend, holidays, reports to write, 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. 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.

The following sections outline some of the current best-practice estimation techniques in software engineering. All these techniques require breaking the project down into more manageable components you can estimate individually. There are other methods of estimating development time. Refer to Appendix A, *References*, for a list of documents that describe these and other estimation techniques in more detail.

Source Lines of Code/Number of Nodes Estimation

Software engineering documentation frequently refers to *source lines of code* (SLOC) as a measurement, or metric, of software complexity. SLOC as a measurement of complexity is popular in part because the information is easy to gather. Numerous programs exist for analyzing textual programming languages to measure complexity. In general, SLOC

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 counts 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. Refer to Chapter 8, *VI Metrics Tool*, for more information on how to use this tool and 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 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 this 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 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 the number of nodes and the time required for development.

Problems with Source Lines of Code and Number 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 dependent on the organization. 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 difficult because of differing levels of experience, different expectations for testing and development methodologies, and so on.
- Size-based metrics are also dependent on the programming language. 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 code is 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 code is 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 that associates 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 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 complement 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 down the project into components that can be more easily estimated. A good guideline is to break the project into tasks that take no more than a week to complete. More complicated tasks are difficult to estimate accurately.

Once you have broken down the project 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 this chapter describes 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 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 keep the estimates anonymous and to have a person outside the development team lead this meeting.

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 developer 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 reexamines and adjusts the estimates. The group then graphs and discusses the new estimates. This process might go on for three or four cycles.

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, refer to 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 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 to produce a number that represents the complexity of the project. You can compare this number 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 on 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 2 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 also.

Estimate project development time separately from scheduling it into your work calendar. Consider estimating tasks in *ideal* person-days, which correspond to 8 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 other responsibilities.

Record your progress at meeting 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 determine 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 each one has clear requirements. To minimize risk, set milestones to complete the most important components first. If, after reaching a milestone, the project falls behind schedule 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 build new development on top of an unstable foundation.

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

A number of the books listed in Appendix A, *References*, provide more information about major and minor milestones.

Responding to Missed Milestones

One of the biggest mistakes people make is to miss a milestone and not reexamine the project as a consequence. After missing a milestone, many developers continue on the same schedule, assuming they will be able to work harder and 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 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 suggestions 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 all the bugs are fixed, or renegotiate the schedule.

Deal with problems as they arise and monitor progress to avoid repeating mistakes or making new ones. Do not wait until the end of the milestone or the end of the project to correct problems.

Missing a milestone should not come as a complete surprise. Schedule delays do not occur all at once. They happen little by little, day by day. Correct problems as they arise. If you do not realize you are behind schedule until the last 2 months of a year-long project, you probably will not be able to get back on schedule.

6

Creating Documentation

This chapter describes techniques for documenting your software.

You should create several documents for software you develop. The two main categories for this documentation are as follows:

- Design-related documentation—Requirements, specifications, detailed design plans, test plans, and change history documents are examples of the kinds of design-related documents 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 is different. Design-related documentation generally is written for an audience with extensive 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. For simple tools that will be used only in-house, you might not need to do 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 a review by the U.S. Food and Drug Administration, 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 different than the format and detail level of an in-house project.

Appendix A, *References*, lists resources that contain information on the types of documents to prepare as part of your development process.

LabVIEW and BridgeVIEW include features that can help you produce some of the documentation you must create. You can use some of the following features of these tools to simplify the process:

- 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 (SCC) tools described in Chapter 11, *Source Code Control Tools*, the SCC tools retain the History window text. 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. You can view this information on screen or save it to a file so you can import it into a word processor to add it to reports. Refer to the *Advanced Features* section in Chapter 11, *Source Code Control Tools*, for more information.
- **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 the names and descriptions of controls and indicators for the VI and the names and paths of any subVIs. You can print this information, generate Web pages, create online help source files, or create word-processor documents.
- Documentation tool—With this tool, you can automate the process of printing documentation for the VIs in your VI hierarchy. Refer to Chapter 9, *Documentation Tool*, for more information.

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 Function and VI Reference Manual* (LabVIEW users) or Appendix A, *HMI 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 later in this chapter. You can use **File»Print Documentation…** to create a printout of a VI in a format 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, Web page, or help file, you can use **File»Print Documentation…** and then select the destination from the **Destination** drop-down menu. If you want to create documentation for multiple VIs all at once, use **Project»Documentation Tool**. Refer to Chapter 9, *Documentation Tool*, for more information.

Documentation for an Application

If you are developing an application for users who are not familiar with LabVIEW or BridgeVIEW, your documentation requires more introductory material. 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 the user starts 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 the user sees documented first. As described in the previous section, you can use the **Print Documentation** dialog box or the Documentation tool to create this documentation.

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 **Print Documentation** dialog box or the Documentation tool to help you create the source material for your help documents.

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 the help files 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) 459-2356; `http://www.blue-sky.com`
- **(Windows)** Doc-To-Help from WexTech Systems, Inc., (914) 741-9700; `http://www.wextech.com`
- **(Macintosh)** QuickHelp from Altura Software, Inc. (408) 655-8005; `http://www.altura.com`
- **(UNIX)** HyperHelp from Bristol Technology, Inc. (203) 798-1007; `http://www.bristol.com`

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. Select **VI Setup** from the pop-up menu and choose **Documentation** from the drop-down menu. In the Help Tag box type the topic you want 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. Now users can access this link from the **Help** window. If the VI is a subVI on another block diagram, you can pop up on the subVI icon and select **Online Help** to link to the selected topic in the specified help file.
- You can use the Help functions from the **Functions»Application Control»Help** palette to link to topics in specific help files programmatically.

VI and Control Descriptions

You can integrate information for the user in each VI you create by using the VI description feature, by placing instructions on the front panel, and by including descriptions for each control and indicator.

VI Description

The VI description in the **VI Information** dialog box from the **Windows»Show VI Info…** menu is often a user's only source of information about a VI. The **Help** window displays a VI description when the user moves the mouse cursor over the VI 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
- Instructions for use
- Descriptions 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 list of important steps is valuable. You might even include a suggestion such as, "Select **Show VI Info…** from the **Windows** 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. Be sure to pop up and select **Data Operations»Make Current Value Default** to save the text when you finish entering the text.

If a text block requires too much space on your front panel, you can include a highly visible **Help** button on the front panel instead. Include the instruction string on its own front panel that pops up when the user clicks on the **Help** button. Use the Window Options in **VI Setup** to configure this help panel as either a dialog box that requires the user to click an **OK** button to close it and continue or as a window the user can move anywhere and close 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»Application Control»Help** palette to open the LabVIEW or BridgeVIEW **Help** window or to open a help file and link to a specific topic.

Control and Indicator Descriptions

Include a description for every control and indicator. You can enter this with the **Data Operations»Description…** pop-up menu item. The **Help** window displays an object description when the user moves 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, and so on)
- Additional information, such as whether the user must set this value always, often, or rarely

Alternatively, you can list the default value in parentheses as part of the VI name. For controls and indicators on the VI connector pane, mark the inputs and outputs as **Required**, **Recommended**, or **Optional**. Refer to the *Connector Panes* section in Chapter 7, *Using Consistent Style: The G Style Guide*, for more information.

7

Using Consistent Style: The G Style Guide

This chapter describes 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.

VI Hierarchy

Organize your VIs in the file system to reflect the hierarchical nature of your software. Make the top-level VIs 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 the VIs for one application and give it a meaningful name, as shown in Figure 7-1. Save the main VIs in this directory and the subVIs in a subdirectory. If the subVIs have subVIs, continue the directory hierarchy downward.

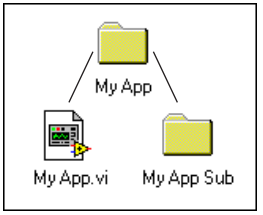


Figure 7-1. Directory Hierarchy

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 and 255 characters on other platforms. Refer to the *Multiplatform Issues* section of Chapter 11, *Source Code Control Tools*, for more information on filename limits for different platforms.

Select **Edit»Preferences…** to make sure the VI Search Path contains <topvi>* and <foundvi>*. The * causes all subdirectories to be searched. In Figure 7-1, `MyApp.vi` is the top VI. This means that the application will search for subVIs in the directory `MyApp` before searching the entire disk. 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 are subject to the 8+3 character limit.

☞ **Note** ***The G Source Code Control tools described in Chapter 11, Source Code Control Tools, do not support LLBs. As described in that chapter in the Using Individual Files Instead of VI Libraries section, you can develop under Windows 95/NT using directories. When it is 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 the **Find File** command of the operating system cannot find them.

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 by selecting **File»Edit VI Library…**. Top-level VIs are listed above and apart from the others in the **File Dialog** dialog box when you select **File»Open**, as shown in Figure 7-2.

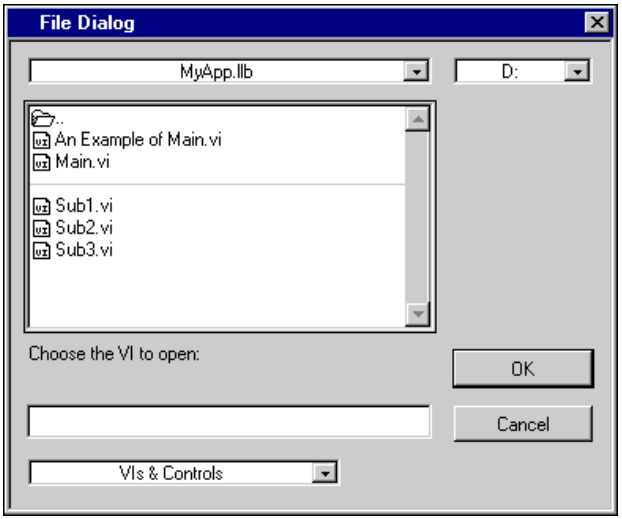


Figure 7-2. Top-Level VIs Listed at the Top of a VI Library

If you use LLBs, use a combination of directories and VI libraries to use 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, as shown in Figure 7-3.

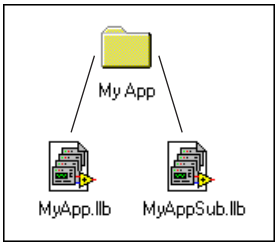


Figure 7-3. Mixture of Directories and VI Libraries

You might move the LLBs that contain 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 a LabVIEW or BridgeVIEW built-in user interface.

A user's first contact with your work, and with LabVIEW or BridgeVIEW, is the front panel, so it had better be high quality.

Consistency

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

Text

Do not be tempted to use all 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 Settings** drop-down menu in the toolbar *without any text or objects selected*. You can select all the labels you need to change and set the font in all of them at once using the **Text Settings** drop-down menu in the toolbar.

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 the pop-up menu. Use carriage returns to make multiline text instead of resizing the text frame.

You can prevent controls and indicators from overlapping because of font changes on multiple platforms by allowing extra space between controls. Fonts are the least portable aspect of the front panel, so always test them on all your target platforms.

Table 7-1 shows suggestions for a consistent set of text styles.

Table 7-1. Examples of Font Styles and When to Use Each

Font	Description of Use
Application Font Bold	Controls and indicators of primary importance.
Dialog Font	Controls and indicators of primary importance; groups of controls or titles; indicators and controls on pop-up panels.
Application Font Plain	Secondary indicators or controls used as constants; groups of controls or titles.

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 bright 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 that cannot display certain color combinations well. For example, black-and-white monitors display black letters on a red background as all black. Use a minimal number of colors, emphasizing 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.
- Use light gray, white, or pastel colors for backgrounds.
- Select bright, highlighting colors only when the item is important, such as an error notification.
- Always check your VI on other platforms and on a black-and-white monitor.
- Be consistent.

Graphics and Custom Controls

You can 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, as shown in Figure 7-4.

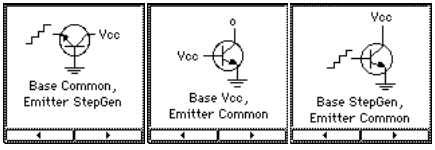


Figure 7-4. Example of Imported Graphics Used in a Pict Ring

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 under Windows or UNIX.

One disadvantage of imported graphics is that they slow down screen updates. The following suggestions might 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. Keep front panels simple to avoid confusing the user. For top-level VIs that users see, place the most important controls in the most prominent positions. Use the **Align Objects** and the **Distribute Objects** drop-down menus to create a uniform layout. Use **Edit»Panel Order…** to arrange controls in a logical sequence. Refer to the *Key Navigation* section later in this chapter for more information. Do not overlap controls with other controls or with their own label, digital display, or other parts unless you are trying to achieve a special effect. Overlapped controls are much slower to draw and might flash. Place any **Start** or **Stop** buttons near the **Run** button on the toolbar for two reasons: The buttons are easier to find, and the **Stop** button will be more prominent than the **Abort** button, if you did not hide it. The user will be 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 the user does not see, you can place the objects 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. Make the window as small 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, 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 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 small part of each. Place front panels that open automatically in the center of the screen by selecting the **Auto-Center** option in the Windows Options version of the **VI Setup** dialog box 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, such as moving a control by one pixel then moving it back, and save the VI or select **File»Save As...** and use the same name.

Controls and Indicators

The following sections guide you on when and how to use various controls and indicators effectively.

Descriptions

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

Labels

The **Help** window displays labels as part of the connector. Label the most important controls and indicators on a front panel in **bold**. 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 Required/Recommended/Optional setting affects the appearance of the inputs and outputs in the **Help** window. The *Connector Panes* section later this chapter describes the Required/Recommended/Optional setting.

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. The following examples is a recommended format for a Boolean where true means reset but the default is false:

Reset Device? (F)

Free labels next to the Boolean can help clarify the meaning of each position on a switch, as shown in Figure 7-5.

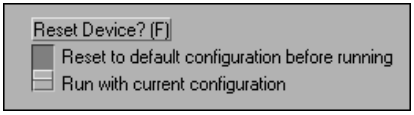


Figure 7-5. Free Labels on a Boolean Control

Enumerations versus 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 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 also will 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 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 in Figures 7-6 and 7-7.

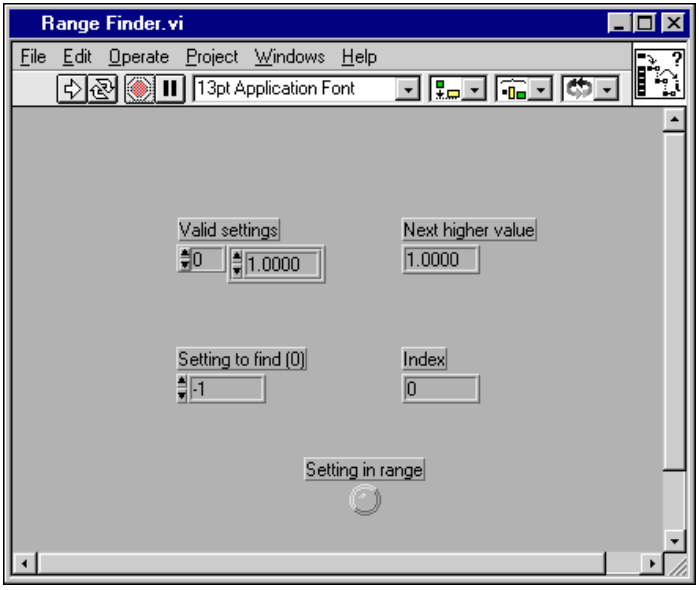


Figure 7-6. Front Panel of Range Finder VI

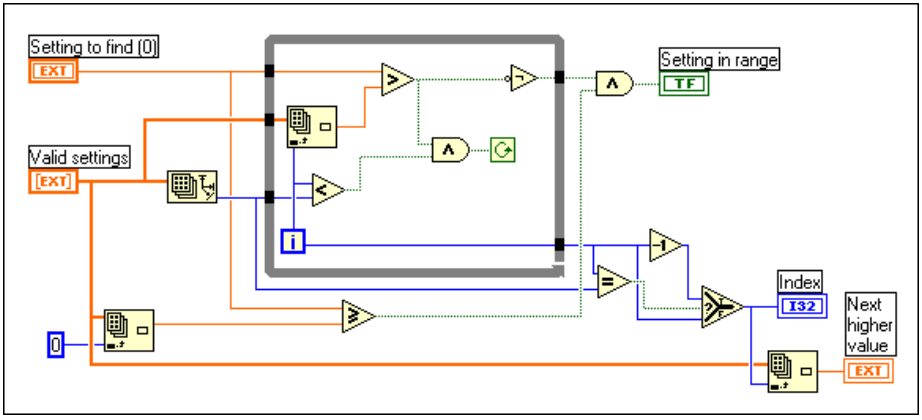


Figure 7-7. Block Diagram of Range Finder VI

Other difficult situations must be dealt with 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 2,000 V for DC but only 1,000 V for AC. If the affected controls like Range and Mode reside in the same VI, put the interlock logic there. Refer to the *Attribute Nodes* and *Local Variables* sections later in this chapter for more information. If one or more of the controls are not readily available, you can request the present settings from the instrument to ensure 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 with default values. Remember to show the default in parentheses in the control label. Do not set default values of indicators like graphs, arrays, and strings without a good reason because that wastes disk space when saving the VI.

Use default values intelligently. In the case of high-level file VIs such as the Write Characters to File VI, the default is an empty path that forces the VI to display a **File Selection** dialog box. 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, including the following suggestions:

- Set the text focus to the main, most commonly used control.
- Dim 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. In general, 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. You can set key navigation options from the **Key Navigation…** item of the pop-up menu of any control. Set the <Enter> key to be the keyboard shortcut to the front panel default control. However, 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 <Esc> 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 opens.

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, using write-to controls with local variables, but the user must be able to override those values by entering his or her own values, using 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 the block diagram. Doing so hides the data flow, making the block diagram more difficult to understand and maintain. Also, using local variables increases the possibility of having *race conditions*, in which multiple locations on the block diagram attempt to modify the same local variable, resulting in the loss of data. Refer to Chapter 28, *Performance*

Issues, of the *G Programming Reference Manual* for more information about using local variables.

VI Setup

To access the **VI Setup** dialog box, pop up on the VI icon and choose **VI Setup**. Think about the window behavior and style of every VI in your project.

In the Execution Options version of the **VI Setup** dialog box, select **Show Front Panel When Loaded** and **Close Afterwards if Originally Closed** for front panels you want to appear and disappear automatically. 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. Refer to Chapter 26, *Understanding the G Execution System*, of the *G Programming Reference Manual* for information on how priorities work.

In the Window Options version of the **VI Setup** dialog box, select **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 toolbar unless the user needs them. You can use the keyboard shortcuts for cut, copy, and paste even if the menu is hidden. Hiding menu bars and using dialog box style makes Help and VI descriptions inaccessible. You can add a **Help** button on the front panel and design it to show the Help window or help file entries programmatically. Remember that you can abort a VI by using the following keyboard shortcuts:

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

Hide the **Abort** button if the user should not abort the VI. Hiding the **Abort** button disables the keyboard shortcut for aborting the VI. You should provide a front-panel Boolean **Stop** button for VIs that loop.

You can hide the single-stepping and execution highlighting buttons to save a small amount of execution time when the VI is finished by turning off the **Allow Debugging** option. However, these debugging tools often are useful to a user trying to understand how the block diagram works.

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 later find that you need another input or output. Changing patterns requires replacing 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 to define data flow. Error in and error out are ideal dataflow connections. If a set of VIs is used together and must be sequenced, you can add a common thread. Refer to the *Adding Common Threads* section later in this chapter for more information.

Position connections for inputs on the left and connections for outputs on the right. This conventional left-to-right data flow prevents complicated, unclear wiring patterns. Figure 7-8 gives examples of good and bad inputs and outputs.

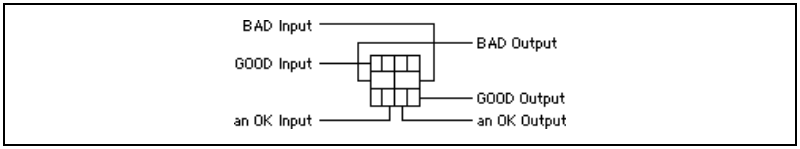


Figure 7-8. Good and Bad Inputs and Outputs

When several VIs use the same inputs and outputs, try to place the inputs and outputs 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/Os are located at the bottom left and right. Placing these inputs and outputs in these locations 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»** from the pop-up menu. If the connector pane is not visible, pop up on the VI icon and select **Show Connector**. 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 in the **Error List** dialog box, the Error window warns 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. This is appropriate for inputs such as refnums, where the VI does not make sense unless the input is wired. You should not make an input required unless it is necessary for the VI to execute properly. Required inputs appear in bold in the **Help** window.

If you make an input optional, the **Help** window does not display it in simple help mode, which helps to simplify the connector pane in the **Help** window. With simple help mode turned off, the input appears dimmed. You should use the optional setting for parameters you rarely need to wire.

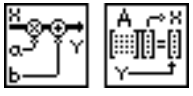
You can specify whether outputs should be recommended or optional, but you cannot mark outputs as required.

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 a language other than English, a well-chosen icon is much more effective.

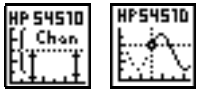
If your VI is a mathematical function, draw a plot of that function, as shown in the following examples.

For simple data-processing functions, depict the input and output data types and the nature of the operation. This can become cryptic, however, so use caution. The following examples are icons for the 1D Linear Evaluation and Solve Linear Equations VIs.



Within driver packages, maintain a unifying theme for groups of icons. Basing your design on drivers for similar instruments makes it easier for

users to convert from one instrument to another with minimal confusion. The following examples are icons for the HP 54570 instrument driver.



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 datalogging VI with a picture of a lumberjack.

Icons for higher level VIs might require some artistic talent. The following icons are examples of good icons.

The Icon Editor offers 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 the Labeling tool to change the font or font size of the text. Some fonts, like Symbol and Glyph, contain many small pictures you can use in your icons. Because it is in bitmap form, text you type 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, so you can create programs that are functional and visually engaging. This section contains recommendations for improving block diagrams in function and in 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 possible. 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:

- Avoid routing wires underneath structures or icons, and never route wires through an icon to a terminal on the other side of the icon.
- Do not use local variables just to avoid having long wires. Every local variable that reads the data makes a copy of it. Using local variables can lead to race conditions. Refer to Chapter 28, *Performance Issues*, in the *G Programming Reference Manual* for more information.
- Reduce the number of pivot points in wires by aligning the source and destination of the wires. Use the cursor keys to remove single-pixel kinks from wires.
- Delete excess wires, such as loops.
- Evenly space parallel wires in straight lines and around corners.

Notice object alignment, consistent spacing, and labels on the long wires illustrated in Figure 7-9.

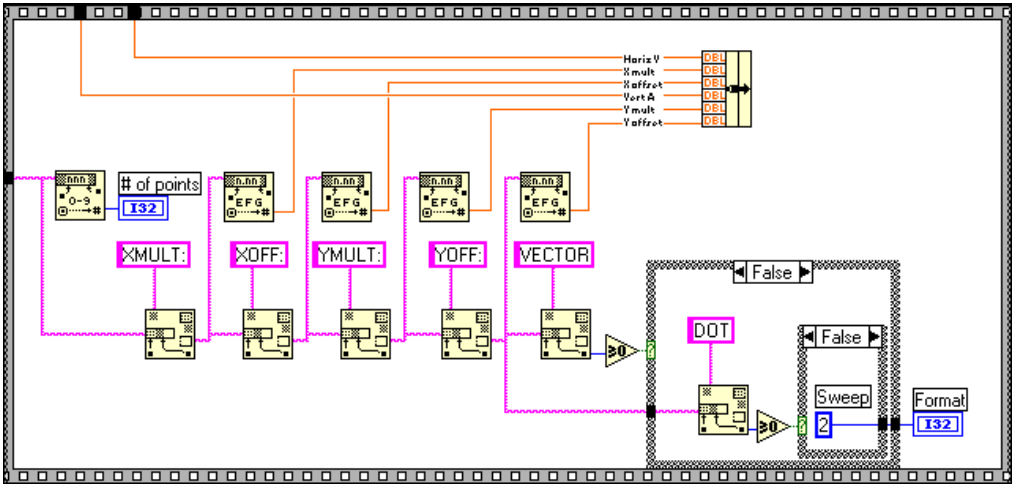


Figure 7-9. Good Wiring in a Simple Block Diagram

Labeling

Give major structures in the block diagram names and descriptions through the **Description…** pop-up menu 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 in the following example:

Paste long comments into small string constants and make them scrollable. Place large scrollable text items off to the side of the block diagram 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, however, do not move to accommodate font changes. Place labels below objects to ensure they stay next to the object even if they grow or shrink on the bottom and right sides. Right-justify any labels you place to the left of an object.

Execution Sequence

The following sections describe programming concepts that will help you take advantage of the natural data flow in block diagrams.

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, or wires, and structures determine execution order.

Data Dependency

As described in Chapter 19, *Structures*, in the *G Programming Reference 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 function, which must perform cleanup regardless of whether an error occurred previously. The Close File function closes the specified file and passes error input as its output. This error information is commonly kept in a cluster that contains a numeric error code, a string that contains 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.

Figure 7-10 shows an example of how the data acquisition VIs use the error cluster. 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.

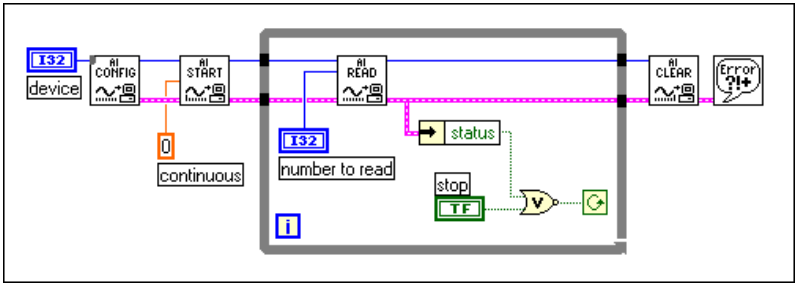


Figure 7-10. Example of How Data Acquisition VIs Use Error Clusters

Sequence Structures

Avoid overusing Sequence Structures. G has a great deal of inherent parallelism. Using a Sequence Structure guarantees the order of execution but prohibits parallel operations. For instance, asynchronous tasks that use I/O devices, such as GPIB, serial, plug-in boards, can run concurrently with CPU-bound operations. *Sequence Structures add no code or execution overhead, but they do restrict parallelism*. Actually, your program might execute faster if you can add parallelism by reducing the use of sequences. Sequences also hide parts of the program and interrupt the natural left-to-right flow of data.

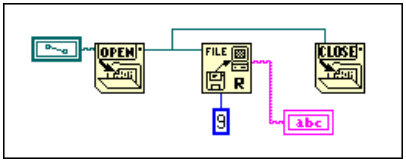
While pure dataflow programming means avoiding Sequence Structures, there are cases where it is appropriate to use them. Use Sequence Structures only if one node must execute before another and cannot be connected by a wire. Refer to the *Data Dependency* section earlier in this chapter for more information.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

Lesson 8, *Additional Topics*, in the *LabVIEW Basics II Course Manual* describes a State Machine, which is an alternative to the Sequence Structure. 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.

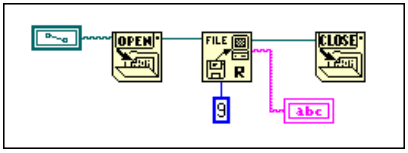
Watch Out for Missing Dependencies

Make sure 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.

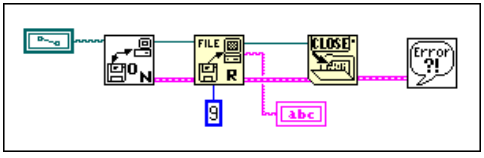
In the following example, there is no dependency between the Read File and Close File functions. More than likely, this program cannot work as expected.



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 that errors might occur. Almost all I/O functions return error information. Make sure your program checks for errors and you deal with them appropriately.

BridgeVIEW and LabVIEW do not deal with errors automatically because users usually want 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
- Broken or improperly working external device or loss of power
- Incorrect file permissions or a lack of disk space

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 can 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 might want the VI to perform the operation regardless of the error it receives.***

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** palette. 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 that contains a numeric error code, a string that contains the name of the function that generated the error, and an error Boolean for quick testing. Figure 7-11 shows how you can use this in your own applications. Notice that the While Loop stops if it detects an error.

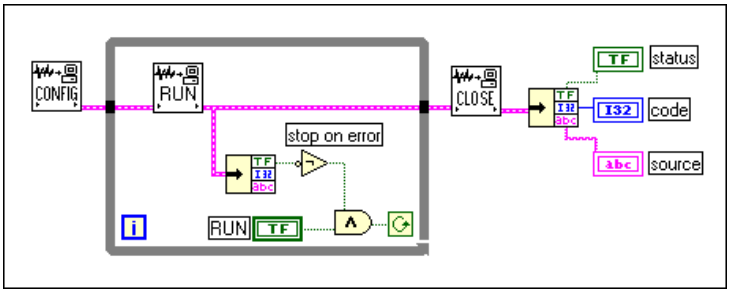


Figure 7-11. Example of How to Use an Error Cluster

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 skill, forethought, and intentions of the programmer. 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 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 the front panel and block diagram do not fit on one screen, place the block diagram in the upper-left corner of the screen. If possible, show the title bar of the front panel and the block diagram. To set the window position, make a small change, such as modifying or moving any object and changing it back, and save or select **File»Save As…** and use the same name.

Figure 7-12 illustrates how a front panel and a block diagram can fit comfortably on a small monitor with room to spare for the **Help** window.

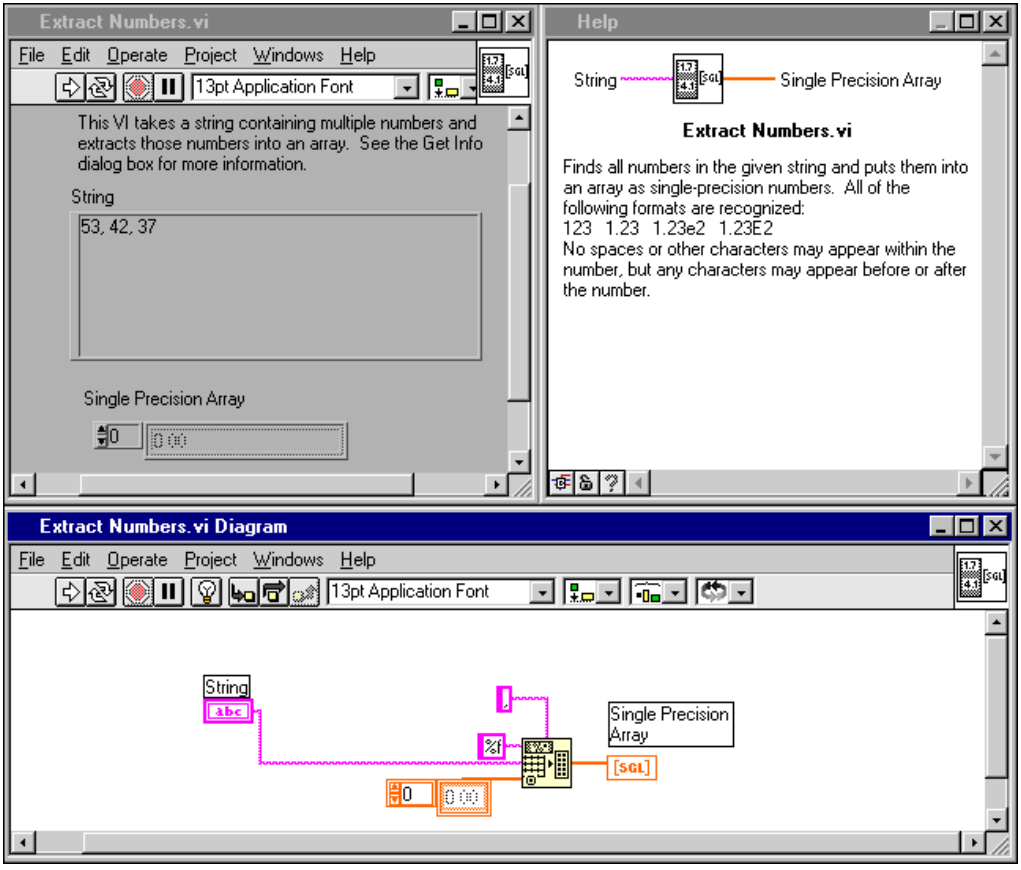


Figure 7-12. Well-Placed Front Panel and Block Diagram

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. Refer to Chapter 28, *Performance Issues*, in the *G Programming Reference Manual* for more information on optimizing G programs.

Code Interface Nodes

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

CIN Description Contents

In the **Description** pop-up menu item of a CIN, or in a scrolling label next to the node, record the following information:

- Source code filename
- Platform and operating system
- Compiler and version
- Location of 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 in the header file with the source code that you enter in the **Description…** pop-up menu item of a CIN. 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. You might want to copy this checklist to use on all 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 **File»Edit VI Library…** to mark top-level VIs.
- ❑ If the VIs will be used as subVIs, use **Edit»Edit Control & 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 backslash (\), slash (/), colon (:), and tilde (~).
- ❑ 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 global variables 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 the **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-and-white icon. Color icons are 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 descriptions 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.
- ❑ Make name label backgrounds transparent.
- ❑ Check for consistent placement of control names, for example, upper left.
- ❑ Use standard, consistent fonts throughout all front panels.
- ❑ Use **Size to Text** for all text for portability and add carriage returns if necessary.
- ❑ Use **Required**, **Recommended**, and **Optional** settings on the connector pane.
- ❑ Put default values in parentheses after input names.
- ❑ Include unit information in names if applicable, for example, Time Limit (10 Seconds).
- ❑ Write descriptions for controls, including array, cluster, and refnum elements. Remember that you might need to change the description if you copy the control.

- ❑ Arrange controls logically. For top-level VIs, put the most important controls in the most prominent positions. For subVIs, put inputs on the left and outputs on the right and follow connector arrangement.
- ❑ Arrange controls attractively, using the **Align Objects** and the **Distribute Objects** drop-down menus.
- ❑ 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 you are using a Boolean for two options, consider using an enumeration instead to allow for future expansion of options.
- ❑ Use Custom Controls or TypeDefs for common controls, especially for rings and enumerations. Include it with VIs.
- ❑ In control VIs, label controls with the same name as the VI, for example, `Alarm Boolean.ctl` has the 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, subVIs, constants, attribute nodes, local variables, global variables, and structures.
- ❑ Add comments. Use object labels instead of free labels where applicable and scrollable strings for long comments.
- ❑ Make comment backgrounds 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 and 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 the left and exit to the right, not the top or the bottom.
- ❑ Align and distribute functions, terminals, and 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 deal with error conditions and invalid values.
- ❑ Show name of source code or include source code for any CINs.
- ❑ Save with the most important or the first frame of structures showing.
- ❑ Review for efficiency, especially data copying, and accuracy, especially parts without data dependency.

Part II

Professional Development Tools

This section of the manual describes the features of the tools.

- Chapter 8, *VI Metrics Tool*, describes how to use the VI Metrics tool to measure the complexity of your application.
- Chapter 9, *Documentation Tool*, describes how to create documentation for VIs in HTML or Rich Text Format (RTF), to create source material for online help files, or to print the material directly to a printer.
- Chapter 10, *VI Comparison Tools*, describes the VI Comparison tools, which you can use to manage different versions of VIs as you develop large applications.
- Chapter 11, *Source Code Control Tools*, describes the G Source Code Control (SCC) tools, which allow you to add files to SCC and access those files from within the LabVIEW or BridgeVIEW environment.

8

VI Metrics Tool

This chapter describes how to use the VI Metrics tool to measure the complexity 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 (SLOC) 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. You also can use those statistics to establish baselines for estimating future projects.

Remember that any metric, such as SLOC, is a crude measurement of complexity. The VI Metrics tool gives you access to several statistics because you might find that some columns are more valuable than others in some cases. For example, you might decide that for user interface VIs you can combine certain statistics to get a better idea of the complexity of a VI. 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 the complexity of a VI.

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

Using the VI Metrics Tool

To use the tool, first open the VI(s) you want to analyze. Select **Project»VI Metrics…**. The **VI Metrics** dialog box appears, as shown in Figure 8-1.

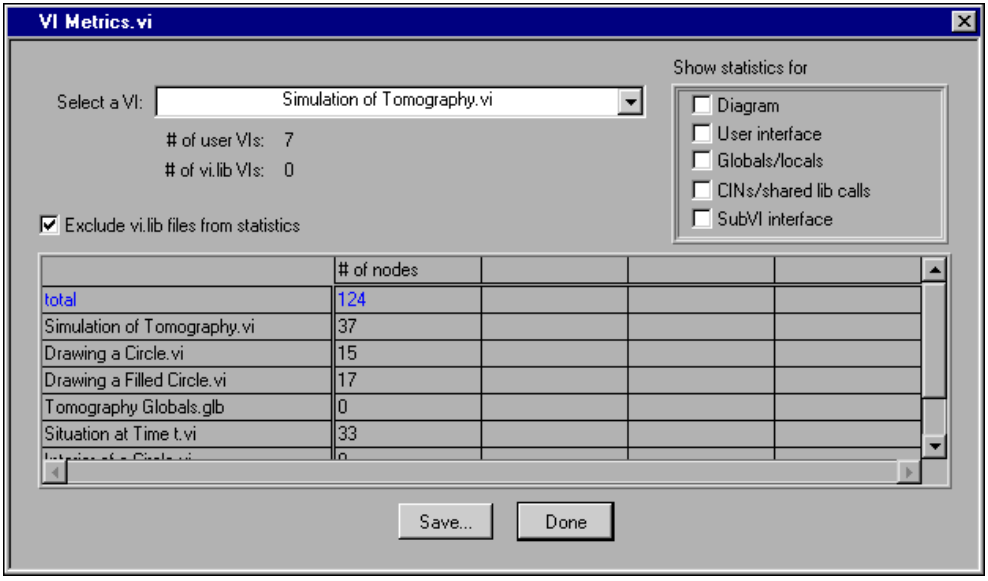


Figure 8-1. VI Metrics Tool Dialog Box

Use the drop-down menu 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 that VI contains. 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 SLOC metric commonly used with textual languages.

The number of nodes includes functions, such as Add, Subtract, and so on, subVI calls, and structures, such as Case, While Loop, and so on. It also includes terminals for front panel objects, constants, local and global references, and attribute nodes. Notice that for attribute nodes, reading multiple attributes with the same node counts as one node. If you want to

know the total number of attributes a VI reads or writes, refer to 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 block diagram in Figure 8-2 contains eight nodes: three terminals, a constant, a random number function, a multiply, a Case Structure, and a For Loop.

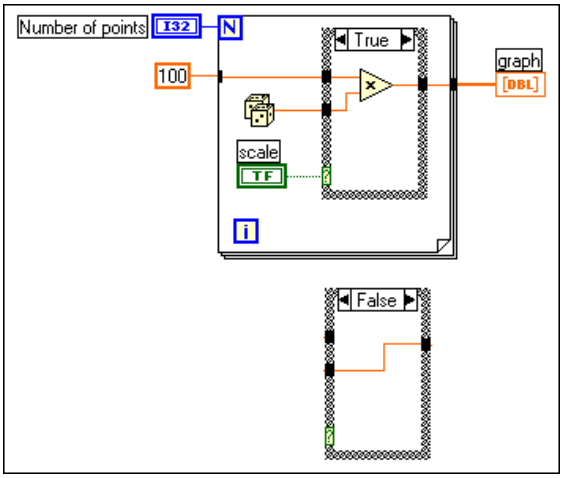


Figure 8-2. Block Diagram with Eight Nodes

Additional Statistics

In addition to measuring the number of nodes, the VI Metrics tool measures 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 right of the **VI Metrics** dialog box.

Block Diagram Statistics

- Structures—Number of For Loops, While Loops, Case Structures, and Sequence Structures.
- 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 diagram depth—Deepest nesting level of block diagrams in a VI. If your VI has no structures, such as cases, loops, and sequences, it has a depth of 0.
- Diagram width—Width of the block diagram in pixels.
- Diagram 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, 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 a single control.
- Indicators—Number of top-level indicators on a VI front panel. A cluster or an array is counted as a single indicator.
- Attribute reads—Number of attribute reads by a VI block diagram. 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. If you write multiple attributes with the same attribute node, each attribute increments this number.

Globals/Locals 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.

CINs/Shared Library Statistics

- CINs—Number of Code Interface Nodes in a VI 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 **VI Metrics** 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 they call and the complexity of those `vi.lib` VIs are not added to the total measurements for the selected hierarchy. This is appropriate if you are trying to get a measurement of the complexity of the code you have written. You can turn off the **Exclude vi.lib files from statistics** 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 by clicking on the **Save…** button in the **VI Metrics** dialog box. Only the columns that are displayed are saved. The information at the top of the dialog box that concerns the number of user VIs and the number of library VIs also is saved to the file. The information is saved in a tab-delimited format so you can easily read it into a spreadsheet or read and parse it using VIs.

9

Documentation Tool

You can use the Documentation tool to create documentation for VIs in HTML or Rich Text Format (RTF), to create source material for online help files, or to print the material directly to a printer.

You can use the **File»Print Documentation…** command to create documentation for a single VI by printing it directly to a printer or by creating source material for online help, Web pages, or word processors.

You can use the **Project»Documentation Tool…** command to create documentation for all the VIs in your applications. The **Documentation Tool** dialog box is shown in Figure 9-1.

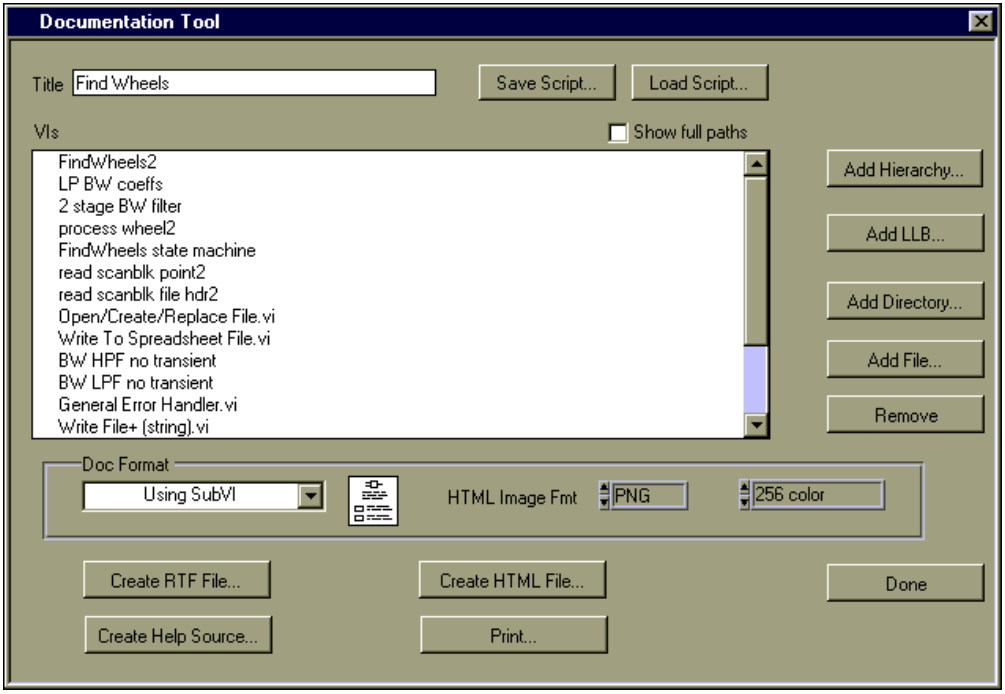


Figure 9-1. Documentation Tool Dialog Box

Use the **Add Hierarchy…**, **Add LLB…**, **Add Directory…**, and **Add File…** buttons on the right side of the dialog box to add VI hierarchies from memory, add all the files in a directory or LLB, or add individual files from disk. The files are listed in the listbox in the dialog box after they are added. The order of the files controls the order that the VIs appear in the resulting documentation. You can reorder the VIs by selecting files from the list and dragging them to the desired location. If the paths are too long to view, you can deselect the **Show Full Paths** checkbox at the top of the dialog box.

When you have the files in the order you want, select one of the following buttons to create the documentation:

- **Create HTML File**—With HTML, you can make your documentation accessible from the World Wide Web. Images are written to external files in `.PNG`, `.JPEG`, or `.GIF` format based on the format you choose in the **Documentation Tool** dialog box. Refer to Chapter 5, *Printing and Documenting VIs*, in the *G Programming Reference Manual* for more information about these formats.
- **Create RTF File**—RTF is a standard format that is supported by a number of word processors, including Microsoft Word.
- **Create Help Source**—This produces an RTF file that is formatted for online help (`.hlp`) files. All images are saved as external bitmaps, and a help project and table of contents are created. These files can then be compiled using a help compiler. Refer to the *Creating Help Files* section in Chapter 6, *Creating Documentation*, for information about compilers for different platforms.
- **Print**—Click this button to print documentation to a printer.

You can use the **Save Script…** button at the top of the dialog box to save the settings and list of files you create to a file. If you click the **Load Script…**button, you can restore the settings from the file you created using the **Save Script…** button.

10

VI Comparison Tools

This chapter describes the VI Comparison tools, which you can use to manage different versions of VIs as you develop large applications. When you make changes to a VI, you might want to compare the VI to an older version to verify the changes you have made. Also, when multiple users are working on the same VIs, you might need to view two versions of the same files to merge changes made by different users.

You can use the following tools to compare VIs:

- **Project»Compare Hierarchies**—Compares two different versions of the same hierarchy of VIs.
- **Project»Compare VIs**—Compares two VIs.
- **Project»Source Code Control»Compare Files**—Compares the local versions of files with the versions under Source Code Control.

Compare Hierarchies

You can use the **Compare Hierarchies** command to compare two hierarchies of VIs. Any file with the same name in both hierarchies is compared. After the comparison completes, **Compare Hierarchies** displays a summary of the differences. You can select a set of VIs that have differences and visually compare them using the Compare VIs tool.

To compare two VI hierarchies, select **Project»Compare VI Hierarchies…**. Use the **Compare VI Hierarchies** dialog box, shown in Figure 10-1, to select two hierarchies to compare.

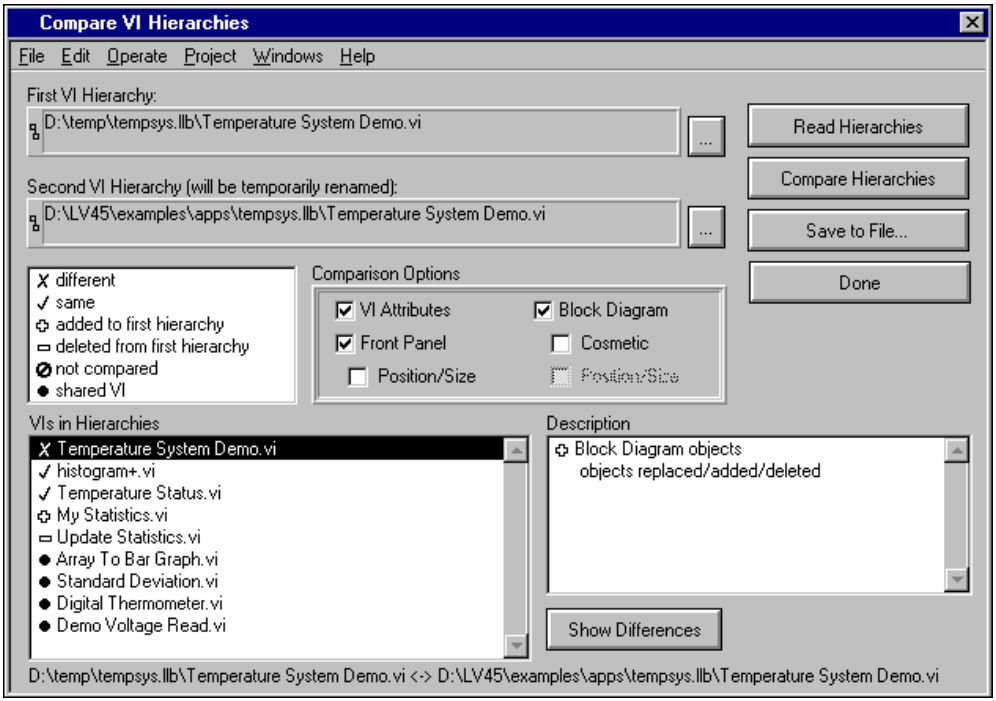


Figure 10-1. Compare VI Hierarchies

Select two VI hierarchies to compare by entering the path to the top-level VI. Use the **…** button to open a file dialog box to select a VI from the file system. You then can use the **Compare Hierarchies** button to compare VIs in the hierarchies. VIs with the same name but different paths are compared and categorized as being the same or different. VIs with the same name and path are considered shared VIs. All other VIs are categorized as added or deleted from the first VI hierarchy. The **VIs in Hierarchies** listbox displays the VIs and a symbol that indicates how they are categorized. As you select different VIs in the listbox, a more detailed description appears in the **Description** listbox to the right. To view the differences on the screen, double click an item in the listbox or click the **Show Differences** button.

The **Read Hierarchies** button categorizes the VIs in the hierarchy. VIs that need to be compared are categorized as `not compared`. You can selectively compare individual VIs by double clicking an item or clicking the **Show Differences** button.

To abort comparisons of large VI hierarchies, use the **Operate»Stop** menu item or press the <Ctrl-.> (Control and the period) key combination.

Comparison Options

You selectively can find differences in the front panel and block diagram of a VI and VI attributes, such as settings in VI Setup. For the block diagram, you can choose to ignore cosmetic differences. A *cosmetic change* is any change that does not affect the execution of the block diagram. Furthermore, you can choose to ignore the position and size changes in front panel and block diagram objects. Position and size changes include movement of an object from front to back and vice versa.

Show Differences

Differences are shown by tiling the front panels and block diagrams of the two VIs in the **Differences** window, as shown in Figure 10-2.

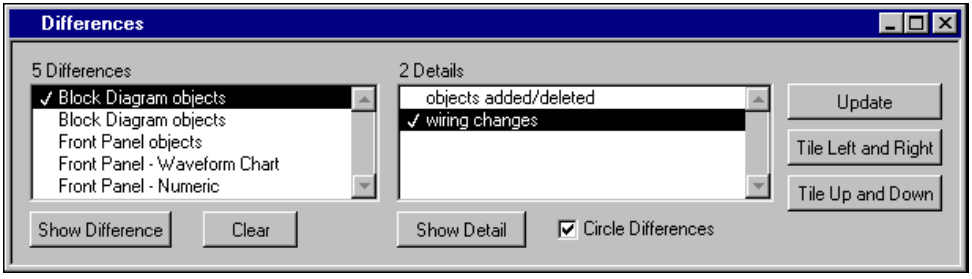


Figure 10-2. Differences Window

The **Differences** window includes a list of differences and details of the selected difference. To highlight a difference, double click an item in the differences list or select an item and click the **Show Difference** button. To highlight a detail, double click an item in the details list or select an item and click the **Show Detail** button. A checkmark indicates the items you selected. You also can use the **Tile Left and Right** and the **Tile Up and Down** buttons to tile the windows of the two VIs you are comparing. Click the **Clear** button to clear the differences list. If you have made edits after a comparison, some of the differences might be stale. Click the **Update** button to again compare the two VIs and update the differences list. You also can show the **Differences** window by selecting **Project»Show Differences**.

When you highlight a difference, objects that are part of the difference are selected. The **Circle Differences** checkbox option allows you to draw a red circle around the object(s) that has changed. Figure 10-3 shows an example of a block diagram difference.

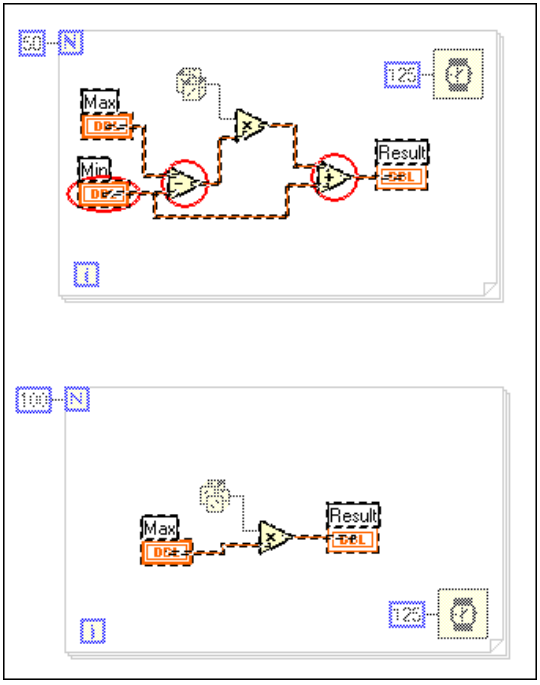


Figure 10-3. Block Diagram Difference

In Figure 10-3, the circled objects are inserted into the block diagram. Objects selected but not circled are not differences. They are selected as *anchor objects* to provide reference points for the difference. Objects that appear dimmed are not part of the difference.

Compare VIs

When developing applications, you might have multiple versions of the same VIs. The Compare VIs feature, also called Graphical Differencing, helps you track changes in your application by comparing multiple versions of a VI. This becomes especially important as your project grows and involves more developers.

You can use the **Project»Compare VIs…** command to graphically compare two VIs. You can select options to control the types of differences you want to detect and view. For example, you can filter out cosmetic changes such as objects being moved or resized. When you compare the VIs, a dialog displays a summary of the differences. If you select an item from the summary, Compare VIs displays and highlights the differences between the two VIs.

To compare two VIs, select **Project»Compare VIs…**. Use the **Compare VIs** dialog box to select two VIs to compare, as shown in Figure 10-4.

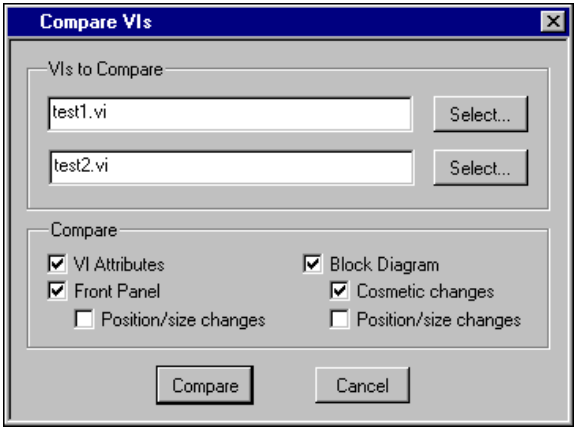


Figure 10-4. Compare VIs Dialog Box

The **Select…** button opens a dialog box to select a VI by name. You can select only VIs that are already loaded into memory.

Comparing very large VIs can be lengthy. You can cancel the comparison of two VIs through the **Comparison Progress** dialog box, shown in Figure 10-5.

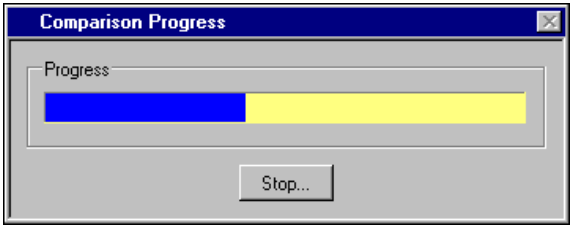


Figure 10-5. Comparison Progress Dialog Box

The progress bar indicates the steps in the comparison algorithm, not the number of differences left to find. When the comparison is complete, the front panels and block diagrams of the two VIs are displayed in the **Differences** window, as shown in Figure 10-2.

Comparison Issues

Because G cannot load two VIs with the same name, you must rename your VIs to compare them. When the Compare VI Hierarchies tool compares two VIs, the first VI loads as is, and the second VI is moved to the temporary directory with a `cmp.` prefix. When using the Compare VIs tool you must rename the appropriate VI to load the two VIs into memory.

However, renaming the VIs does not affect the name of subVIs. Because the Compare VI Hierarchies tool does not rename the subVIs, *the renamed VI will link to the subVIs loaded by the first VI.*

Source Code Control»Compare Files

You can use the **Source Code Control»Compare Files…** command to compare files from projects under Source Code Control with the local versions of those files.

To compare VIs under SCC, select **Project»Source Code Control»Compare Files…**. You can use the **SCC Compare Files** dialog box, shown in Figure 10-6, to compare the project files on the master directory with the project files on the local directory.

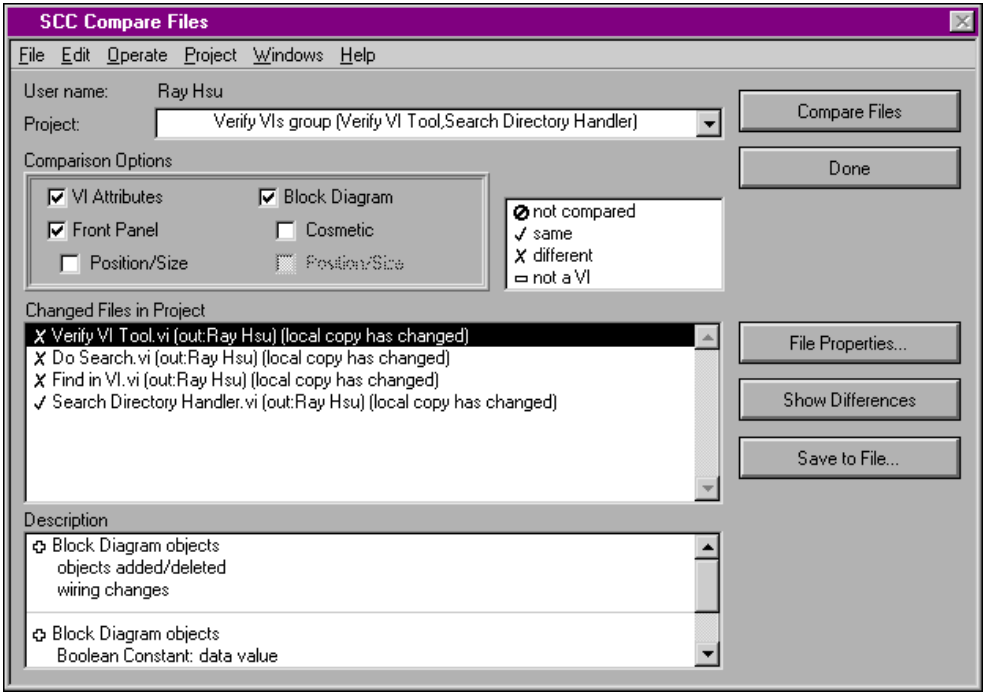


Figure 10-6. SCC Compare Files Dialog Box

After you select a project, the **SCC Compare Files** dialog box displays all changed files between the master copy and local copy. You then can use the **Compare Files** button to compare the changed files. Only VI files are compared. Other external files are not compared. This tool is similar to the **Project»Compare VI Hierarchies…** tool. Refer to the *Compare VIs* section for more information.

11

Source Code Control Tools

This chapter describes the G Source Code Control (SCC) tools. The 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 by allowing you to share files among multiple developers and multiple projects. SCC tools maintain a centralized master copy of project files. As you make changes, you update the 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 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 the new version of the file in to Source Code Control. The file 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 you can 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. They also can keep track of the changes made to those documents. The ability to track the evolution of software is important to most organizations that are concerned with quality.

Using Individual Files Instead of VI Libraries

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 an LLB, only the VI library name itself needs to be limited to 8+3 characters in length.

You should not use VI libraries for files 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. You would have to check out the entire LLB if you wanted to make changes to any file within the VI library. The G SCC tools do not support LLBs because VI libraries do not permit fine enough control over individual VIs.

The primary reason you might want to use VI libraries is that you need to support Windows 3.1, where filenames are 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 develop under other operating systems the G SCC tools support, such as Windows 95/NT. When developing, save your VIs as individual files, not in LLBs. When you need to release your VIs for use on Windows 3.1, you can save them into LLBs at that point.

The File Manager tool can help you move VIs to and from VI libraries.

QuickStart Guide to Using the SCC Tools

Following is a brief summary of the steps required for configuring and using the SCC tools:

1. **Administrator sets up system**—At least one user in your workgroup should be selected to administer the SCC system. This user should select the **Administration** option when installing the Professional Developers Tools.

   The administrator is responsible for initializing the SCC system and setting up systemwide preferences. You do this by selecting **Project»Source Code Control»Administration…**. When you set up the SCC system there are a few decisions you need to make:

   – **Which storage system do you want to use?** The SCC tools can manage files or they can integrate with a third-party tool. Refer to

the following *Selecting the Source Code Control System* section for more information.

– **Will users be developing on multiple platforms or a single platform?** This decision might help decide the storage system you will use, as well as how to configure that system. For multiplatform development, National Instruments recommends the built-in system. For single platform development you can use either the built-in system or one of the third-party SCC tools that are supported.

Once the system is up and running, there is typically not a lot of work involved on the part of the administrator unless you want to change access privileges for files or limit access for specific users.

If you are using ClearCase, refer to the *ClearCase* section later in this chapter for more information.

2. **Users configure the SCC tools to access the system**—The main decision users must make is where they will have a *working directory.* The working directory is a directory on your local system where all VI development takes place. As you retrieve files from SCC, they are retrieved to this directory and its subdirectories. You cannot add files to SCC that are outside of this directory.

   With the built-in system, it is important to understand the difference between the working directory, which is where you do your work, and the master directory, which is where SCC stores the latest versions of files. Users will not modify the master directory except when they check in files or create projects.

   With SourceSafe, the tools take care of copying files from the SourceSafe database to your working directory or vice versa.

   With ClearCase, your work directory is the same directory as the master directory. ClearCase uses a virtual file system model that allows multiple users to have a unique view and set of files in the same directory. Refer to the *ClearCase* section later in this chapter for more information.

3. **Users set up VIs for use with the SCC tools**—As mentioned above, all files that the user will work with must be in the working directory or one of its subdirectories. In addition, files that will be stored under SCC cannot be stored in LLBs. LLBs are not flexible enough for them to work well with an SCC system because they do not allow for easy access to individual files. You can select **Project»File Manager…** to convert your LLBs to directories of VIs.

   In converting LLBs to directories, you might find that you have some files with names that cannot be managed by the file system (for

example, it contains a separator character such as :, \, or /). Consequently, you might have to rename some files and update references within your hierarchy.

If you need to support Windows 3.1, refer to the *Cross-Platform Source Code Control* section later in this chapter for suggestions on the best strategy for handling development.

4. **Users create SCC projects**—A project in the SCC tools corresponds to a hierarchy of VIs. To create a project, first open the top-level VI of a hierarchy. Then select **Project»Source Code Control»Project…** and click the **New Project…** button.

   The **SCC Edit Project File List** dialog box lets you select your VI hierarchy, which can then be added to SCC. As mentioned previously, all files that you want to add to SCC must be in your working directory or one of its subdirectories. They cannot be in LLBs.

   You can add other project-related files such as documentation and shared libraries by using the **Extra Files…** button in this dialog box.

5. **Users use SCC commands to access project files**—At this point, you can begin to work with files under SCC. All commands related to SCC are in the **Project»Source Code Control** menu. Following is a brief description of the remaining commands:

   – **Check Files In**—Use to copy a file from your working directory to SCC so that other users can access it.

   – **Check Files Out**—Use to check out files to your working directory so that you can modify them.

   – **Compare Files**—Use to compare the files in your working directory with those under SCC.

   – **Retrieve Files**—Use to copy files from SCC to your working directory.

   – **Advanced**—Use to create reports, view the history of files under SCC, access previous versions of files, and delete files.

Selecting the Source Code Control System

The G Source Code Control tools can integrate with third-party Source Code Control systems. You can have the SCC tools manage the source internally using the built-in system, or you can have the tools communicate with a third-party system that is responsible for storing, retrieving, checking in, and checking out files. The SCC user interface in LabVIEW and BridgeVIEW is the same regardless of whether you use the built-in system or a third-party system.

Currently, two third-party SCC systems are supported: Microsoft Visual SourceSafe for Windows 95/NT and Rational Software ClearCase for Solaris 2, as shown in Figure 11-1. National Instruments might add support for other third-party tools in the future based on user demand.

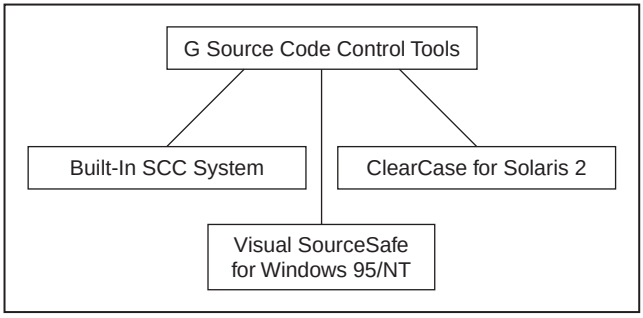


Figure 11-1. G SCC Tools Can Work with Built-In and Third-Party Systems

In some cases, the decision to use one of the third-party tools might be made based on the fact that your company has standardized on one of these tools. If not, you must decide which strategy you want to use for managing your files.

Built-In System

The built-in system manages files by copying them to and from a shared network directory that all users in a development group can access. It has many of the capabilities of full-featured Source Code Control systems, including access to previous versions of files, history information maintenance, and labeling files for easy retrieval.

The built-in system has some advantages over using third-party systems, including the following:

- It is available for all platforms except Windows 3.1. Most third-party tools work only for a single operating system.
- Most third-party tools have additional licensing costs.

However, there are some disadvantages to the built-in system compared to a third-party system, including the following:

- The built-in system does not provide as much security as most third-party systems. You can control access somewhat by using file system permissions, but you do not have the same level of control as with a more powerful SCC system.
- The built-in system does not use compression for files on the server. Some third-party systems can store files on the server and differences between versions, or deltas, in a compressed format. However, because VIs are binary files, most Source Code Control systems do not handle deltas for VIs efficiently.

Third-Party SCC Systems

In addition to the advantages and disadvantages described in the previous section, there are some additional points to notice when considering a third-party SCC system. Although you can use a third-party SCC system, you might not be able to take advantage of all the features that tool offers. For instance, the G SCC tools do not support a feature called branching, where different projects use different versions of a VI. If you want to branch development, you need to make a copy of the VIs you want to change.

If you already have VIs stored under an existing SCC, the G SCC tools will not be aware of them. You will need to create projects for the files using the G SCC tools.

With Visual SourceSafe, the G SCC tools have to keep track of the modification date and check-out status for files internally. Consequently, you should not modify VI files or check them out from the SourceSafe Explorer because the G SCC tools will get out of sync with the state of the files. Instead, if you need to perform any operation that modifies a VI file in the SourceSafe database, you need to do that modification using the SCC tools in LabVIEW or BridgeVIEW.

Administrator Setup

The administrator has an extra menu option, **Project»Source Code Control»Administration…** that he or she uses to set up the SCC system so others can use it.

The administrator uses the **SCC Administration** dialog box, as shown in Figure 11-2, to select and configure the SCC system the G SCC tools use for maintaining files. The drop-down menu lets him or her choose from the built-in system or from other supported third-party systems. Refer to the *Selecting the Source Code Control System* section earlier in this chapter for information on the differences among these options.

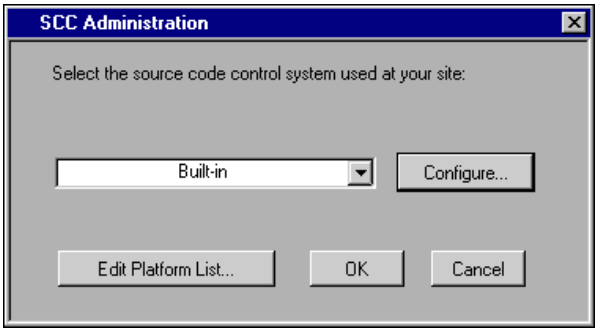


Figure 11-2. SCC Administration Dialog Box

Configuring the SCC System

The following sections describe how to configure the built-in SCC system, Microsoft Visual SourceSafe for Windows 95/NT, and ClearCase for Solaris 2.

Built-In System

If you select the built-in system, the **Administer Builtin System** dialog box, as shown in Figure 11-3, allows you to configure systemwide options that affect all users. If you need to change one of these settings later, click the **Configure…** button in the **SCC Administration** dialog box.

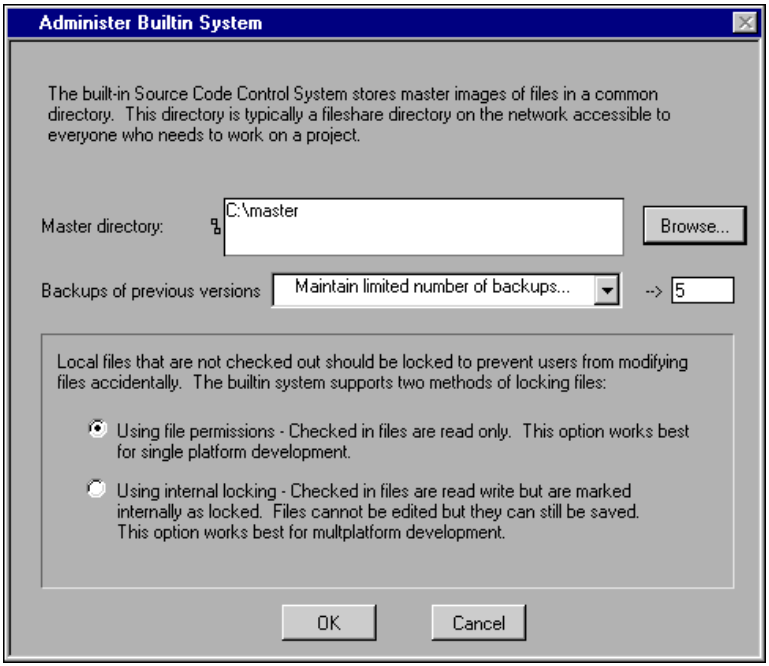


Figure 11-3. Administer Builtin System Dialog Box

You must select a master directory, which is where the Source Code Control tools store the files you add to SCC. Its important that all users have access to this directory. If you want users to be able to modify files, they need to have read-write access to this directory. 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 available disk space.

In addition, under Windows 95/NT, each user should connect to the volume that contains this directory and map that volume as a network drive, for example, as `D:\` rather than as a UNC path of the form `\\machine\volume`. UNC filename support varies slightly between Windows 95/NT. Consequently, the SCC tools do not support UNC filenames.

Use the drop-down menu in the center of the **Administer Builtin System** dialog box to configure how many backups to maintain for files under Source Code Control. If you decide to maintain file backups, a copy of the old file is created when a user checks in a file. You can configure the system so 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 you can retrieve older versions of the files. If you choose to have no limits on backups, you 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 by selecting **Project»Source Code Control»Advanced…**.

You have a set of options for deciding how files are locked for each user. Locking prevents users from accidentally modifying files without first checking them out. The **Administer Builtin System** dialog box gives you the following options:

- **File System Locking**—For many sites, this is the appropriate selection. With **File System Locking**, as you check files in they are marked as read-only in the file system. As you check them out, they are changed to read-write. This works well for most developers.
- **Internal Locking**—With this option, as you check files in they are locked by LabVIEW or BridgeVIEW. If you open the **Show VI Info** dialog box, you will see that the lock option is turned on. When you check files out they are automatically unlocked. This is the best choice for multiplatform development, for example, Windows, Macintosh, and/or UNIX development. When you bring a VI from another platform, it must be recompiled and saved. With internal locking, you can actually save the VI. The internal locking just prevents you from making edits to a VI. The SCC tools can distinguish between real changes and simple modifications such as a recompile that occurs when you load the VI.

Visual SourceSafe

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

After you install Visual SourceSafe, select **Project»Source Code Control»Administration…** to display the **SCC Administration** dialog box. Select Microsoft Visual SourceSafe from the drop-down menu and click the **OK** button to add configuration files used by the G SCC tools to

the Visual SourceSafe database. The G SCC tools maintain the following two files:

- `scclist.lst`—This is a master file list 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 users can select for retrieving files.

You also can configure the platforms, such as Windows 95, 68K Macintosh, and so on, that a user can select. Refer to the *Edit Platform List (Advanced Option)* section later in this chapter for more information.

ClearCase

The Professional Developers tools provide the interface for using ClearCase in the G development environment on Solaris 2. The administrator must create a ClearCase Versioned Object Base (VOB) and an associated storage space outside of G before users can use Source Code Control with ClearCase. When the administrator sets up the Source Code Control system from inside G, he or she must provide the VOB directory as the master directory. Because ClearCase uses a virtual file system, users also must use this VOB directory as their local directory.

Once the administrator and users configure their ClearCase systems, ClearCase is invisibly integrated into the G Source Code Control interface.

Take the following steps to complete the administrator setup for ClearCase:

1. Install ClearCase if it is not already on your system. Refer to the ClearCase documentation for instructions.
2. If you want to create public VOBs, set up a registry password or locate the existing registry password.
3. Create the VOB.
    a. Set your `umask`. For shared VOBs, the suggested `umask` setting is `2`.
    b. Create a mount point directory, such as `/vobs`.
    c. Create or choose a directory to store the VOB database, such as `/vobstore`.
    d. Type `cleartool mkvob -public -tag/vobs/vobname /vobstore/vobname.vbs`, where `vobs` is the mount point directory and `vobstore` is the directory where the VOB database is stored.

e. Enter registry password (from step 2) if creating a public VOB.

f. Enter comments, if any.

g. Restore your `umask` to its original setting.

4. Mount the VOB. For example, type
`cleartool mount /vobs/vobname`
5. Provide the name of the VOB (for example, `/vobs/vobname`) to users.
6. For ClearCase, you must specify that VIs are binary files and decide if you want to compress files. For each file extension that your VIs use, you must define an association of these extensions with their file type. For most G applications, this includes VIs (`.vi`), controls (`.ctl`), and globals (`.glb`). If you use any other extensions, you must define file types for those as well.
   a. Choose a file type for VIs, such as `file`, `compressed_file`, or `binary_delta_file`. You also can define your own file type and type manager. Refer to the ClearCase documentation. National Instruments recommends `compressed_file` because `binary_delta_file` storage does not work as well.
   b. You specify this association in a *magic file* in one of the paths in the *magic path*. Refer to the ClearCase documentation to determine where the magic path is.
   c. Somewhere in the search path for magic files, create a file with a `.magic` suffix.
   d. Add the line: `compressed_file : -name "*.vi" ;` Replace `compressed_file` with your chosen file type.

      For controls, add the line: `compressed_file : -name "*.ctl" ;`

      For global variables, add the line: `compressed_file : -name "*.glb" ;`
7. Set up a view. Refer to the *ClearCase* section later in this chapter for instructions on setting up a view.
8. Set the view and launch LabVIEW or BridgeVIEW.
9. Select **Project»Source Code Control»Administration…**.
10. Select **ClearCase** as the Source Code Control system.
11. The **ClearCase Configuration** dialog box opens.

12. Set VOB directory (e.g. `/vobs/vobname`).
13. Click **OK** in the **ClearCase Configuration** dialog box.
14. Click **OK** in the **SCC Administration** dialog box.

Optional Administrator Setup

You must complete these steps if you want to add VIs to ClearCase.

1. Complete local configuration. Refer to the *ClearCase* section later in this chapter for more information.
2. Set your `umask` to `2` to give other users in your group access to any new directories you create.
3. Set a view and launch LabVIEW or BridgeVIEW.
4. Copy files to the VOB directory. Use **Project»File Manager…** or copy them using UNIX commands. The File Manager tool is useful because it can convert LLBs to VIs. LLBs are not supported by the SCC tools.
5. Open the top-level VI for a new project.
6. Create a new project using **Project»Source Code Control»Project…**.
7. Click the **New Project…** button.
8. Select the top-level VI from the pop-up menu near the top of the dialog box. It will automatically assign a name for the project. You can change the assigned name. Click **Save**.
9. The VI hierarchy and its subVIs are now in Source Code Control. Repeat this process for additional hierarchies.

Edit Platform List (Advanced Option)

Clicking the **Edit Platform List…** button in the **SCC Administration** dialog box displays the **SCC Edit Platform List** dialog box, as shown in Figure 11-4.

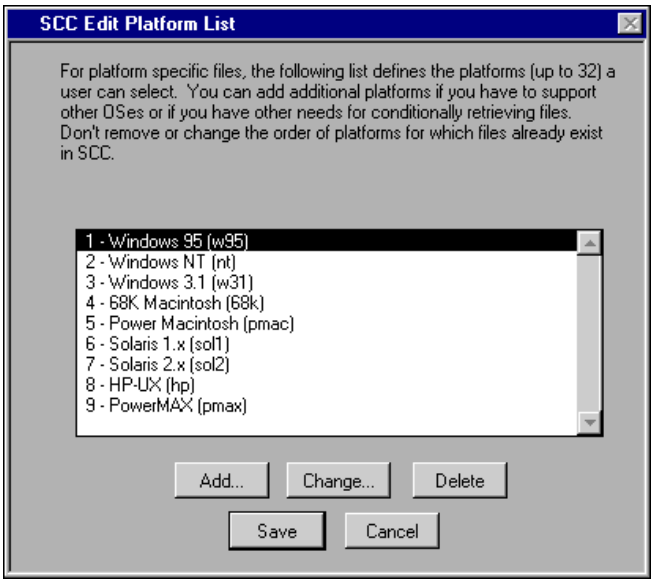


Figure 11-4. SCC Edit Platform List Dialog Box

This dialog box contains a list of the platforms users can select from when they perform local configuration. In general, most users should not change any of the options in this dialog box.

Each entry consists of a long name and an abbreviation. The user sees long names 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 **SCC 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 need to support only a single platform. Assuming you do not change any existing items, the G SCC tools automatically detect the platform being used. If you delete one of the existing platforms and you later decide you want to add support for that platform, you must add that

name back in exactly as it was spelled originally to have the auto-detection feature 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, not by name.

Although these tools do not directly support Windows 3.1, the platform is listed as an option so you can have files specific to Windows 3.1. Refer to the *Multiplatform Issues* section later in this chapter for more information.

Local Configuration

Each user has to configure the G SCC tools before they can use them by selecting **Project»Source Code Control»Local Configuration…**.

The main thing each user needs to do in the **SCC Local Configuration** dialog box, as shown in Figure 11-5, is select the SCC system the administrator configured. Ask your SCC administrator to determine which system your site uses.

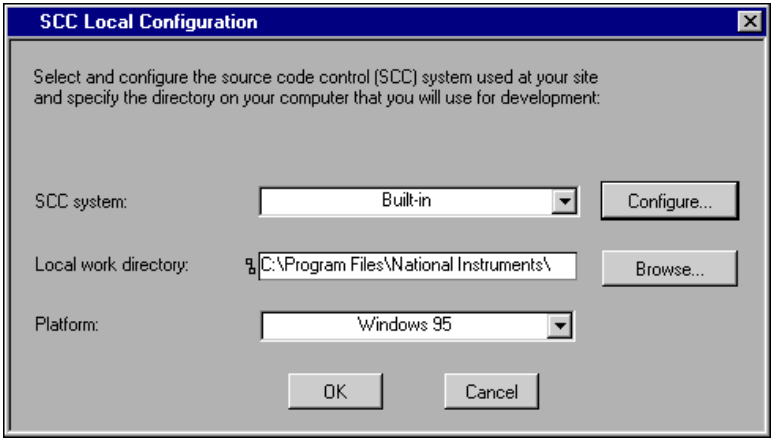


Figure 11-5. SCC Local Configuration Dialog Box

Configuring the SCC System

Users need to choose the SCC system they are using, set the local work directory, and set the platform option in the **SCC Local Configuration** dialog box.

Built-In System

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

Select **Project»Source Code Control»Local Configuration…** to display the **SCC Local Configuration** dialog box. Select **Built-in** from the **SCC System** drop-down menu to display 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 click the **Configure…** button in the **SCC Local Configuration** dialog box to access this dialog box.

After you configure the master directory, you need to configure the local directory as described in the *Local Work Directory* section later in this chapter.

Visual SourceSafe

To use Visual SourceSafe, relatively little configuration is needed. Install Visual SourceSafe and make sure your administrator assigned you an account and password. In the **SCC Local Configuration** dialog box, specify the local work directory and the platform as described in the *Local Work Directory* section later in this chapter.

ClearCase

In addition to providing the correct local directory, users must set up a ClearCase view and make sure it is set every time they launch LabVIEW or BridgeVIEW. To emulate the model of having local copies of files, users must set up unique labels to mark current working versions of files. When working under this model, users must edit their views to show labeled files before most recent versions. Users must take the following steps to configure their ClearCase Source Code Control system:

1. Find out the VOB directory name from the administrator(s). For example, `/vobs/vobname`.
2. Create a view.
   a. Set `umask` (typically `2` for shared files). Refer to the ClearCase documentation for more information.
   `cleartool mkview -tag` *`viewname viewstorage`*
   (for example, `cleartool mkview -tag george /viewstore/george.vws`)
   Each user should have his or her own view.

   b. Restore `umask` to original value.
3. Set your view. For example, `cleartool setview` *viewname*.
4. Choose a label to mark current working versions of files. Your userid in all uppercase letters is a good choice. Your label must be unique among all who will access a particular VOB.
5. Edit your view to show versions with this label.
   a. Edit your view. With your view set, execute `cleartool edcs`.
   b. Using the text editor (launched by `cleartool`), add the line: `element * LABELNAME`, replacing *LABELNAME* with your chosen label to the view configuration spec just after the line `element * CHECKEDOUT`.
6. Launch LabVIEW or BridgeVIEW. You must set the view every time you launch LabVIEW or BridgeVIEW.
7. Select **Project»Source Code Control»Local Configuration…**.
8. Choose **ClearCase** as your Source Code Control system.
9. Enter the VOB directory name given to you by the administrator in step 1.
10. Enter the label name from step 4.
11. Click **OK**.
12. Enter your local working directory, which must be the same as the VOB directory.
13. Click **OK**.
14. To freeze your working set of VIs so that changes made by other users do not affect your work until you retrieve them, select **Project»Source Code Control»Retrieve Files…** and retrieve the files you want to freeze. When you select **Retrieve Files…**, G indicates if the files in your view are out of date and if retrieving files will update your view by showing a snapshot of the latest versions of those files.

Local Work Directory

The local work directory you set in the **SCC Local Configuration** dialog box is the directory where you store all your work. The idea of the work directory is that all users will have the same set of subdirectories and files within that directory. As you change files within your directory and check them in to Source Code Control, other users can copy the files from Source Code Control to their own work directory.

The exact location for the work directory is completely up to you, unless you are using ClearCase. You should make sure you have enough disk

space on the drive that contains that directory because it needs to be able to contain all the files you work on that are under Source Code Control.

Platform Drop-Down Menu

You select the platform you are working with in the **Platform** drop-down menu in the **SCC Local Configuration** dialog box. Unless your administrator has changed the setup, the G SCC tools should correctly detect the platform you are using automatically. In general, you will not need to change this setting. Refer to the *Multiplatform Issues* section later in this chapter 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

The following sections outline how to create and update a Source Code Control project, how to add or delete files from a project, and how to work with project groups.

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, which is a VI and all or some of its subVIs. In addition, an 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 on a VI hierarchy, the G SCC tools can help you 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 update the project.

Managing Multiple Hierarchies

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 create a project group to make it more convenient to access the VI hierarchies simultaneously. Refer to the *Project Groups* section later in this chapter for more information on project groups.

Creating a Project

To create and edit the contents of projects, select **Project»Source Code Control»Project…** to display the **SCC Project** dialog box, as shown in Figure 11-6.

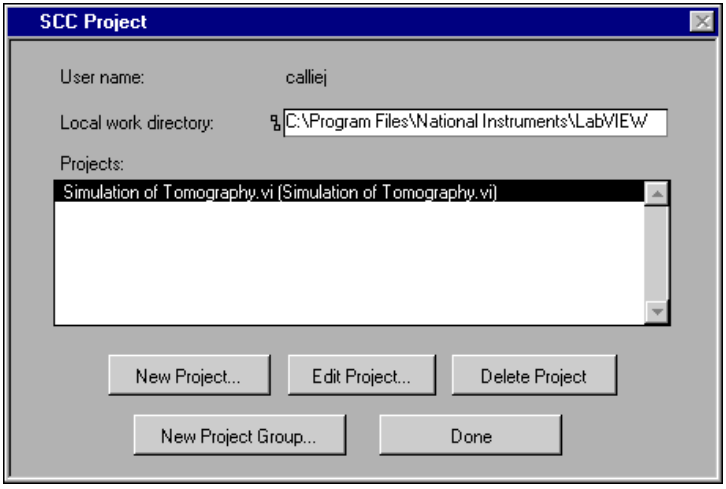


Figure 11-6. SCC Project Dialog Box

A Source Code Control project consists of a hierarchy of VIs, which is a VI and all or some of its subVIs. To create a project, open the top-level VI you want to add to SCC. Select **Project»Source Code Control»Project…**. Click the **New Project…** button in the **SCC Project** dialog box. In the **SCC Edit Project File List** dialog box, select the top-level VI for which you want to create a project, as shown in Figure 11-7. The project contents listbox updates the list of subVIs the VI calls.

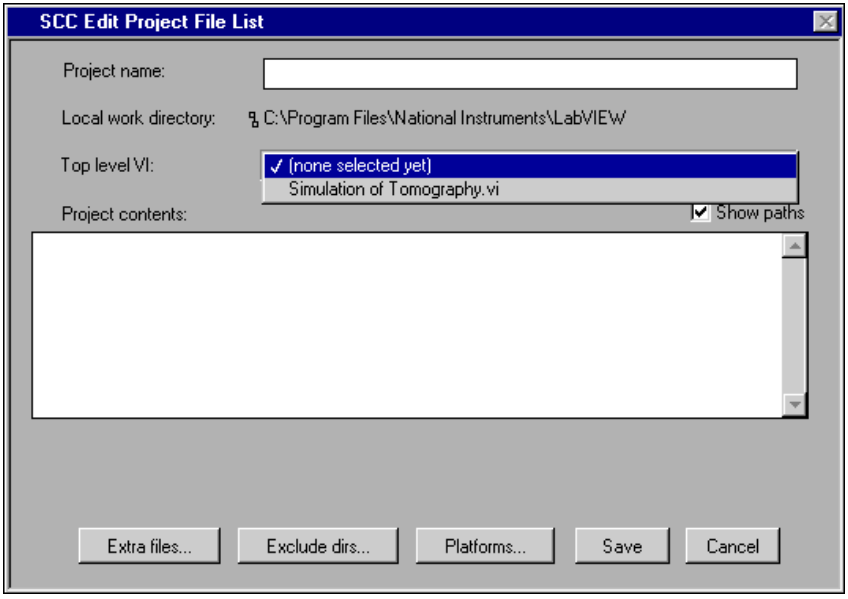


Figure 11-7. Edit Project File List Dialog Box

If you click the **Save** button, 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. Refer to the *Checking Out Files* section later in this chapter for more information.

The files in your hierarchy must be present in your working directory or in an LLB before you can add those files to Source Code Control.

You can add files that are not part of your hierarchy to the project by clicking the **Extra Files…** button. You can add project-related documents such as proposals, specifications, and illustrations to SCC. You also can 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. Refer to the *Adding Extra Files to a Project* section later in this chapter for more information.

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

The **Platforms…** button lets you work with platform-specific files. If your application calls Code Interface Nodes (CINs) or DLLs, you might have

files that are specific to a given operating system. A DLL might be available under Windows but not other operating systems. If you write a CIN for multiple platforms, you need a different variant of the VI that contains the CIN for each platform. The **SCC Edit File Platforms** dialog box deals with both issues because it lets you mark a file as platform specific and lets you create variants of a file for different platforms. Refer to the *Multiplatform Issues* section later in this chapter for more information.

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

Updating a Project

As you develop your VIs, you 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 and then select **Project»Source Code Control»Project…**. Select the project in the **SCC Projects** dialog box and click the **Edit Project…** button.

As always, the files must be in your working directory and cannot be in LLBs unless they are in one of the excluded directories for the project. By default, files in `vi.lib` are excluded.

SCC File Wizard

When you edit a project using **Project»Source Code Control»Project…**, the SCC tools compare your hierarchy to the version that is already under SCC. If there are differences in the list of files, the SCC File Wizard walks you through the process of updating SCC to reflect local changes.

This wizard first presents a dialog box that summarizes the differences between the local version of your hierarchy and the version that is under SCC. This wizard detects whether one hierarchy has files that the other does not and whether files are in different locations in the local hierarchy from the hierarchy under SCC. If you choose to edit the project, a series of dialog boxes describe these differences in more detail and give you the option to change the project or ignore the difference. No changes are made unless you click **Save** in the final dialog box, which summarizes the new list of files.

Managing Files with the Same Name

In general, you should be cautious when working with files with the same name because it is easy to accidentally link the wrong version of a file to a hierarchy. The SCC tools detect if you accidentally link the wrong version of a file when you edit a project using the **SCC Edit Project File List** dialog box. You have the choice to avoid changing the project by moving the existing file in SCC to the new location or by adding the new file as file with the same name in a different location.

Removing Files from a Project

To remove files from a project, select **Project»Source Code Control»Project…**. Select the project in the **SCC Project** dialog box and click the **Edit Project…** button. Select the file(s) in the **SCC Edit Project File List** dialog box and click the **Remove** button or double click 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. A checkmark appears 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 you want to exclude. You can access the **Exclude Directories** dialog box by clicking the **Exclude Dirs…** button in the **SCC Edit Project File List** dialog box. Files you exclude in this fashion are dimmed 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 for this is that multiple projects can share files. Also, even files that are not part of a project are retained in SCC because they might be important for historical reasons. To permanently delete a file, select **Project»Source Code Control»Advanced…**. Select the file you want to delete and click the **Delete File** button in the **SCC 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 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, select **Project»Source Code Control»Project…**. Select the project in the **SCC Project** dialog box and click the **Edit Project…** button. In the **SCC Edit Project File List** dialog box, click the **Extra Files…** button to display the **SCC Edit Extra Files** dialog box, as shown in Figure 11-8.

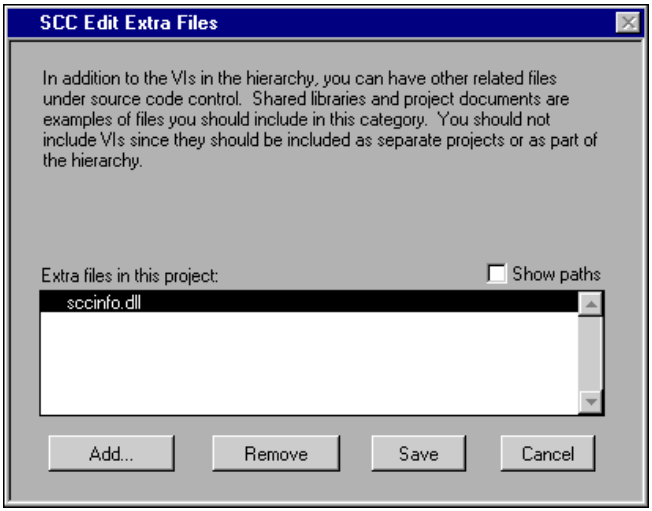


Figure 11-8. Edit Extra Files Dialog Box

Add or remove extra files by selecting them and clicking the **Add…** or **Remove** buttons. Once you are finished, click the **Save** button to commit any changes you make.

Do not use the **SCC Edit Extra Files** dialog box to add VIs to a project. 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. Refer to the following *Project Groups* section for more information.

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 might have 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 considered 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 you want the project group to contain. Select **Project»Source Code Control»Project…** and click the **New Project Group…** button in the **SCC Project** dialog box to display the **SCC Edit Project Group** dialog box, as shown in Figure 11-9.

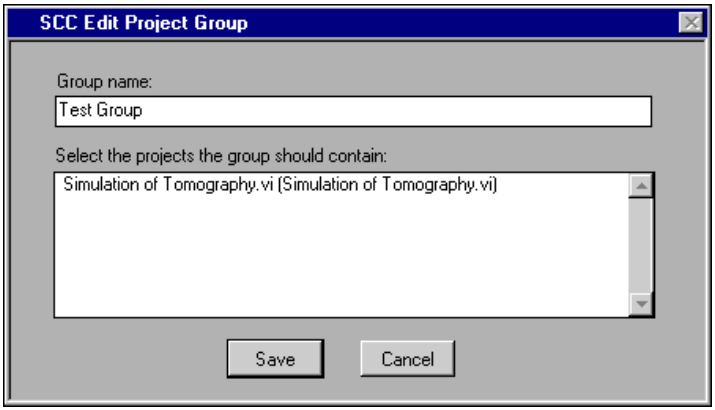


Figure 11-9. Edit Project Group Dialog Box

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

Once you create a project group, its name shows up as a project you can select when you use the **SCC Check Files Out**, **SCC Check Files In**, **SCC Retrieve Files**, or **SCC Advanced** dialog boxes.

Accessing Files

The following sections show you how to retrieve, check out, and check in files of a Source Code Control project.

Retrieving Files

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

The **SCC Retrieve Files** dialog box lets you select a project or project group from which to retrieve files, as shown in Figure 11-10.

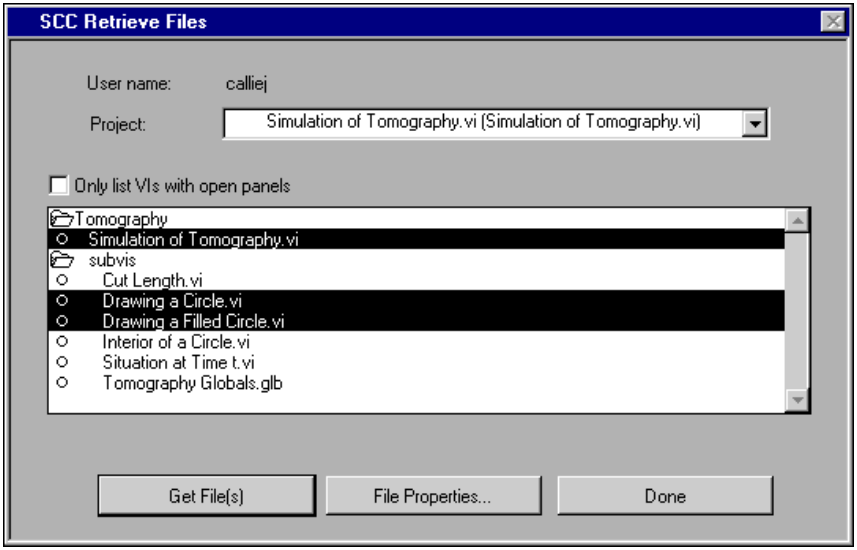


Figure 11-10. SCC Retrieve Files Dialog Box

To retrieve a file, select it and click the **Get File(s)** button. To retrieve multiple files, shift click items to select them and then click the **Get File(s)** button. Because the list can be long if you have several files in your project, you can use the **Only list VIs with open panels** checkbox to ignore unopened files.

File Status

The **SCC Retrieve Files** dialog box automatically compares the local copy of files in the project with the master copy 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 out the file or replace the local copy with the version from the server. You should not 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, refer to the *Updating a Project* section earlier in this chapter.

- Server copy does not exist—Either the file has been deleted on the server by another developer or it is a new file you have created. If it is a new file you created and you want to update the project, refer to the *Updating a Project* section earlier in this chapter.

File Properties

Clicking the **File Properties…** button in the **SCC Retrieve Files** dialog box displays the **SCC File Properties** dialog box, as shown in Figure 11-11. This dialog box gives you a summary of information about the file, including the projects it belongs to, the checkout status, and modification dates.

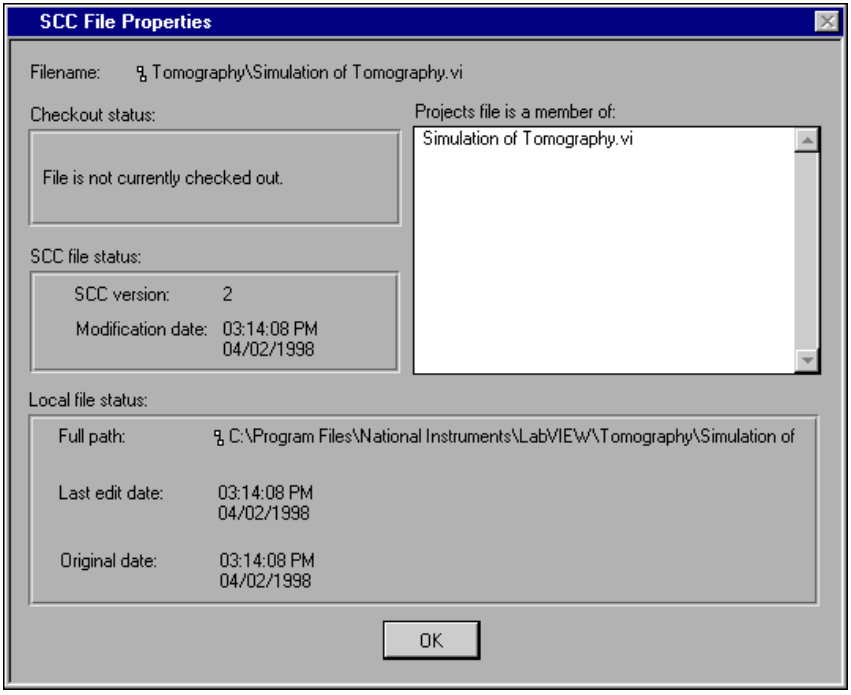


Figure 11-11. SCC File Properties Dialog Box

You can get more information about a file, including a file history, by selecting **Project»Source Code Control»Advanced…**.

Checking Out Files

When you want to modify a file, first check out the file to reserve it so 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 to your local system. It is then unlocked so you can edit the VI. While you have the file checked out, nobody else can check out the file or modify it. This helps to ensure that only one developer at a time modifies a VI.

To check out files, select **Project»Source Code Control»Check Files Out…** to display the **SCC Check Files Out** dialog box, as shown in Figure 11-12.

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

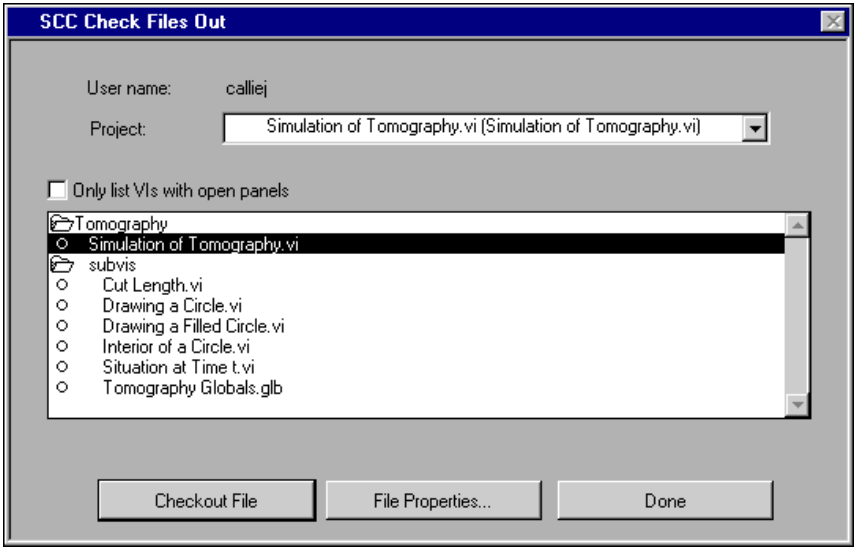


Figure 11-12. SCC Check Files Out Dialog Box

The interface for checking out files is almost identical to that for retrieving files. You cannot check out files that are already checked out to you or to another developer. If the file is checked out, the list indicates the username of the developer with the VI. If you need to check out multiple files, shift click items in the listbox to select the files you want to work on and click the **Checkout File** button.

In general, you should avoid checking out files for long periods of time. Instead, you should try to make incremental changes to your files. When you are sure your files are in good shape, you should check them in. Whenever you check in a file, make sure you have tested it thoroughly so you do not cause problems for other developers. If you need to modify other VIs before you can check in a specific VI, check out the other VIs, make the changes, and test the VIs before checking in any of the VIs.

If you need to make several changes to a VI, consider checking in the file between modifications and then check the file back out to start the next modification. Not only does this allow other developers access to your changes, but it also gives you a checkpoint 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 to a VI, select **Windows»Show History…** to enter notes about your changes. You can check out a file for several days. The History window helps you 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 out the file.

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 older version and you need to distinguish between two different versions. You can create reports and access old versions by selecting **Project»Source Code Control»Advanced…**.

Checking In Files

When you finish making changes to a file you have checked out, select **Project»Source Code Control»Check Files In…** to display the **SCC Check Files In** dialog box, as shown in Figure 11-13. This dialog box lets you copy your version of the file 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.

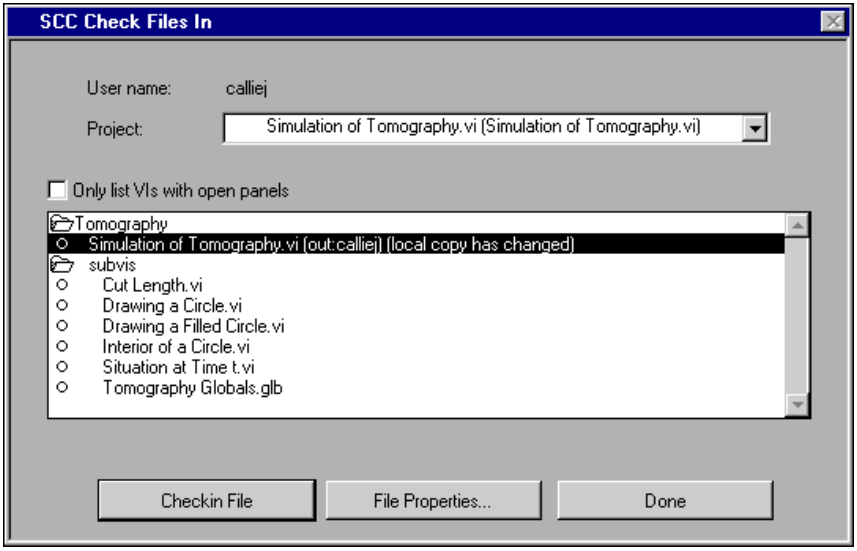


Figure 11-13. SCC Check Files In Dialog Box

The interface for checking in files is similar to that for checking out files and for retrieving files. You can check in a file only if it is checked out to you. Your username 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 made. If you used the History window to record changes, the **SCC Edit Change Comments** dialog box initially contains the history text since you checked out the file, as shown in Figure 11-14.

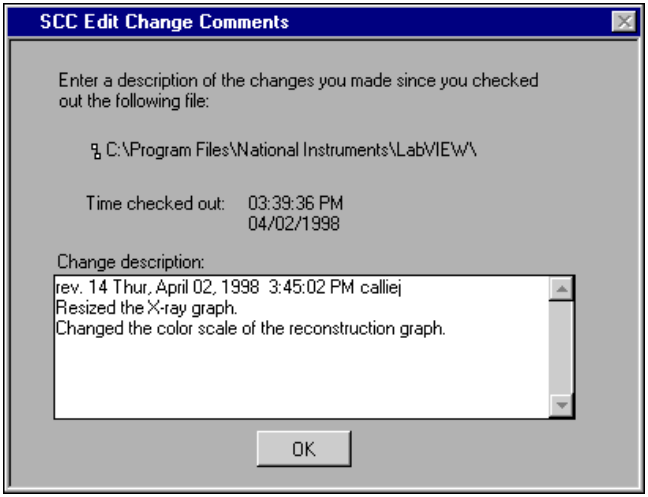


Figure 11-14. Edit Change Comments Dialog Box

You can edit this text to expand the information or to 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 older version and you need to distinguish between two different versions. You can create reports and access old versions by selecting **Project»Source Code Control»Advanced…**.

SCC User Name

When you check in files or modify projects, the G SCC tools use your LabVIEW or BridgeVIEW username to access the Source Code Control system. You can change this username by selecting **Edit»User Name…**. Also, you can control whether to prompt for a username when you launch LabVIEW or BridgeVIEW by selecting **Edit»Preferences…** to display the **Preferences** dialog box. Select the appropriate options in this dialog box.

It is important that the username 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. If more security is important, consider using an alternative SCC system such as Microsoft Visual SourceSafe or Rational Software ClearCase for Solaris 2 for storing files. Visual SourceSafe and other third-party tools prompt you to enter a password whenever you access files or modify projects. Refer to the *Selecting the Source Code Control System* section earlier in this chapter for more information on the built-in system and third-party systems.

Advanced Features

You can access most advanced features by selecting **Project»Source Code Control»Advanced…** to display the **SCC Advanced** dialog box, as shown in Figure 11-15. This dialog box contains features for viewing all files under SCC, determining the projects those files belong to, accessing older versions of files, permanently deleting files, and creating reports.

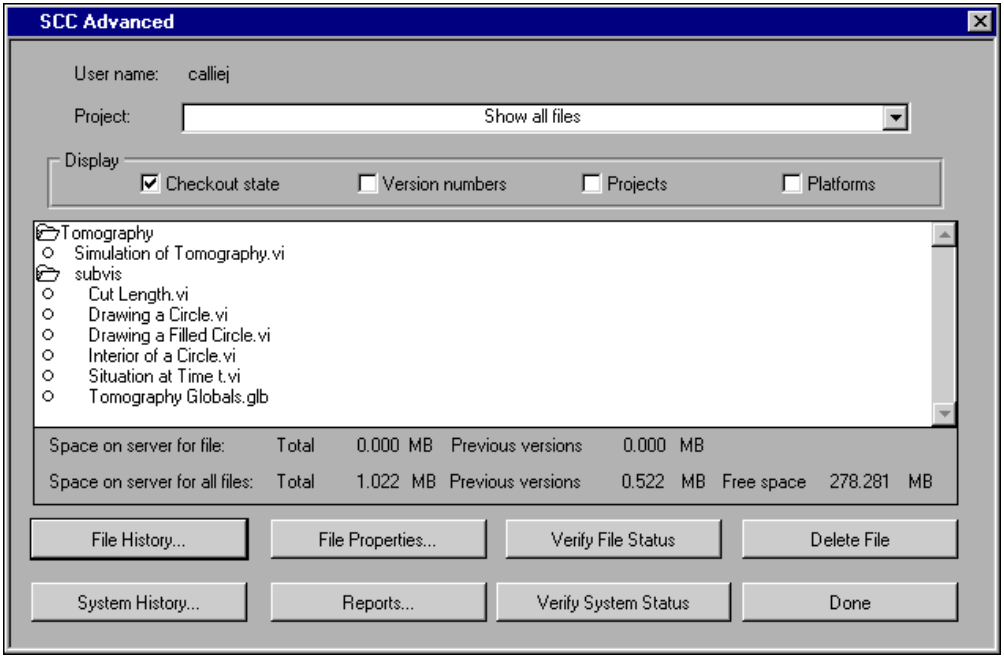


Figure 11-15. SCC Advanced Dialog Box

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 although a file might not be needed currently, it is important historically in terms of being able to understand the evolution of your software. If you later decide you need to retrieve an older version of the project, the file is still present in SCC.

If you decide you do not need a file, you can delete it by selecting **Project»Source Code Control»Advanced…**. In the **SCC Advanced** dialog box, select the files you want to delete and click 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 the SCC system maintains and the history log for that file. Also, you should enable the **Display Projects** option in the **SCC Advanced** dialog box to verify that the files you want to delete are not used currently by any other projects.

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. The comments you enter when you check in a file are added to the log for a file.

To view the SCC history information for a file, select **Project»Source Code Control»Advanced…**. Click the **File History…** button in the **SCC Advanced** dialog box to display a scrolling list of the SCC history for the file, including dates and names for every person who modified the file.

It is important to understand the difference between the SCC file history and the VI history you modify with a VI History window. You can use the VI History window as a place to enter notes about changes to a VI as you make them. This VI history is a part of the VI. If you give the VI to someone else, the VI history is still present unless you reset it using an option from the History window. When you check in a file, regardless of whether it is a VI or another document, such as a specification, you are given a chance to make an entry in the SCC file history. For VIs, the default text for this entry is the VI history text since you checked out the file. However, you are given a chance to change this entry so 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 **SCC View File History** dialog box appearance depends on 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, most systems should allow you to access previous versions of files. Refer to the *Accessing Previous Versions of Files* section later in this chapter for more information. Also, the built-in system, and some third-party systems such as Visual SourceSafe and ClearCase for Solaris 2, gives you the option to label the current version of a file so it is easier to recognize if you later want to retrieve it. For example, you might label a file as beta so you can easily retrieve that checkpoint version later. Refer to the *Labeling Versions of Files for Easy Retrieval* section later in this chapter for more information.

It is important that you enter detailed comments when you check in files and if you use the History window because that information can help you understand the evolution of your software. It also is extremely helpful if you need to determine which version introduced a problem.

If you want to view a summary of transactions for multiple files, click the **System History…** button instead of the **File History…** button in the **SCC Advanced** dialog box. Refer to the following *System History* section for more information. If you use the report generation feature of the **SCC Advanced** dialog box, you can save the system and file histories for files in a project. Refer to the *Creating Reports* section later in this chapter for more information.

System History

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

As with the **SCC View File History** dialog box, the **SCC View System History** dialog box appearance depends on 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, most systems allow you to label the current version of all files in projects so they are easier to recognize if you later want to retrieve them. For example, you might apply a beta label to all files in a project or multiple projects so you can easily retrieve those versions later. Refer to the *Labeling Versions of Files for Easy Retrieval* section later in this chapter for more information.

Accessing Previous Versions of Files

Most SCC systems 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 also is useful if you give a version of a file to a customer, continue development, and subsequently need to retrieve the same version you sent the customer so 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 might 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 the SCC system so it maintains only a limited number of previous versions of each file. In this case, as you check in a newer version of a file, a fixed number of previous versions will be maintained.

If you label versions of files with the built-in system, the labeled versions are not automatically deleted and are not counted as part of this system administrator limit. Labeled versions are maintained until you choose to delete them.

With the built-in system, you can access previous versions of files from the **SCC View File History** and the **SCC View System History** dialog boxes. The **SCC View File History** dialog box allows you to retrieve a previous version of a single file. The **SCC View System History** dialog box allows you to scan the system for all versions of a file with a specific label and allows you to retrieve those versions. This can be useful in taking a snapshot of your product that you can retrieve 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 Visual SourceSafe and ClearCase for Solaris 2, 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, Visual SourceSafe and ClearCase for Solaris 2 maintain history for files.

Labeling Versions of Files for Easy Retrieval

With the built-in SCC system, the **File History…** button in the **SCC Advanced** dialog box lets you retrieve older versions of files. By default, however, the names you see are based on the older version number and the version creation date. The **SCC View File History** dialog box lets you optionally label the current version so 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 to maintain. Labeled versions of a file are not deleted unless you delete them yourself, either from the **SCC File History** dialog box or from the **SCC Advanced** dialog box.

If you use 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 company provides. Also, notice that with third-party tools, the way in which previous versions are maintained and when they get deleted is configured using the tools provided with the third-party SCC system.

If you use the built-in SCC system and you need to label multiple files, you can use the **SCC View File History** dialog box on each one, but that can be tedious. Instead, if you want to take a snapshot of all the files in a project and apply the same label to all the files, use the **SCC View 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

An important feature of any SCC tool is the ability to generate reports that describe system and file activity 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 **SCC Advanced** dialog box lets you create basic reports that describe 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 **SCC Reports** dialog box depend on the SCC system you use.

Built-In System

If you use the built-in system, you get the dialog box shown in Figure 11-16.

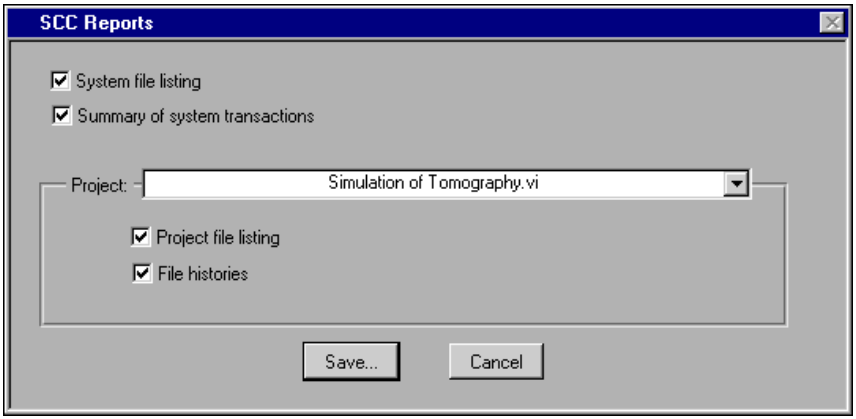


Figure 11-16. SCC Reports Dialog Box for Built-in SCC System

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

The **Summary of system transactions** option gives the same information as in the **SCC View System History** dialog box. It lists all check outs and check ins, project creations, file creations, and file deletions. It includes the username, the date, and a brief summary of the changes for each transaction.

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 **SCC View 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 and all information about subsequent changes to the file, including the username, the date, and a brief summary of the changes.

Visual SourceSafe

With Visual SourceSafe, you can generate the file listings from the **SCC Reports** dialog box. You have to generate transaction listings and other reports from the Visual SourceSafe environment. Remember that the

system history can be viewed by looking at the history of the `sccfiles.lst` file. Refer to the *System History* section earlier in this chapter for more information.

Multiplatform Issues

This section describes multiplatform issues, such as cross-platform SCC, filename limitations, and platform-dependent SCC files.

Cross-Platform Source Code Control

Unlike most Source Code Control tools, the built-in SCC system lets you access files from all the platforms LabVIEW and BridgeVIEW support except Windows 3.1. Refer to the *Using Individual Files Instead of VI Libraries* section earlier in this chapter for more information.

Windows 3.1 is not supported because the SCC tools manage VIs as individual files and the Windows 3.1 8+3 character naming limitation makes this impractical. While LLBs make it possible to use longer names, they do not provide the level of transparent access to files that is needed for the built-in SCC tools.

Although you cannot use the SCC tools on Windows 3.1, National Instruments understands that customers need Windows 3.1 support. As a developer, you might need to deploy applications to customers that are using systems with Windows 3.1. In that case, develop your application under Windows 95/NT. On those platforms you can use individual files for VIs. When you are ready to distribute to Windows 3.1 users, you can save copies of your VIs into LLBs. An easy way to convert your files to or from LLBs is to use the File Manager tool, which allows you to convert between directories and LLBs.

Once the administrator has set up the Source Code Control system, you can access it from any supported platform. There are some issues you should be aware of if you are developing VIs on or for multiple platforms, including filename limitations and platform-dependent SCC files.

Filename Limitations

Macintosh, Windows 95/NT, and UNIX systems each have restrictions about filenames and paths 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 the following characters: \, :, /, *, ?, ", < , >, and |. Paths are limited to 255 characters, including the filename.

Windows NT supports FAT and NTFS file systems. FAT filename restrictions are the same for Windows 95. NTFS filenames are limited to 255 characters in length and have 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/NT, you should avoid paths that are deeply nested, longer than 255 characters. The 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 variety of computing platforms. You can open most VIs on any platform LabVIEW and BridgeVIEW support, and they will run without modification. You can use the G SCC tools 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 might 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 have the file treated as platform specific so that it is not normally retrieved on unsupported platforms.
- 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 DLLs, the libraries apply only to specific platforms. The VIs that call the libraries are platform independent, assuming 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

You can mark a file as platform specific by selecting **Project»Source Code Control»Project…** to display the **SCC Project** dialog box. Select a project that contains the file you want to modify and click the **Edit Project…** button to display the **SCC Edit Project File List** dialog box. Click the **Platforms…** button to display the **SCC Edit File Platforms** dialog box, as shown in Figure 11-17. This dialog box lets you mark the platforms for which a file is available.

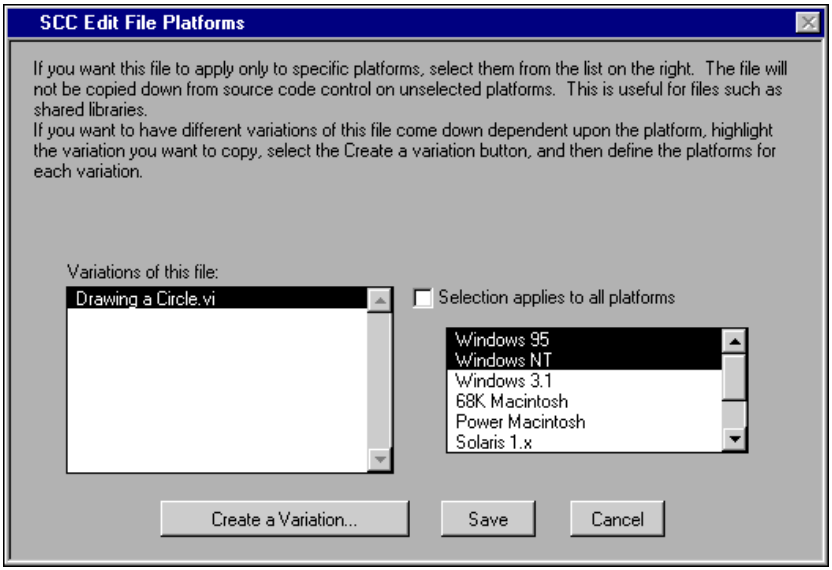


Figure 11-17. SCC Edit File Platforms Dialog Box

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

Each user of the G SCC tools will have configured the platform for which they want files by selecting **Project»Source Code Control»Local Configuration…**. Assuming you do not modify this list, the **SCC Local 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, select **Project»Source Code Control»Project…** to display the **SCC Project** dialog box. Select a project that contains the file and click the **Edit Project…**button to display the **SCC Edit Project File List** dialog box. Click the **Platforms…** button to display the **SCC Edit File Platforms** dialog box. To create a new variant, click the **Create a Variation…** button. Choose the new variant and select the platforms to which it applies. If necessary, modify the existing variants to ensure there is no overlap. The **SCC Edit File Platforms** dialog box does not allow you to apply multiple variants to the same platform.

Retrieving Files for a Different Platform

In some cases, you might 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, reload the CIN, and 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 prone to error. Instead, a better method is to check out the VI on each platform, replace it with the VI with the correct block diagram, recompile, and reload the CIN. Afterward, you can check in the resulting VI as the variant for the current platform.

The **SCC Check Files In**, **SCC Check Files Out**, and **SCC Retrieve Files** dialog boxes restrict the list to the current platform. To change the platform, select **Project»Source Code Control»Local Configuration…** to display the **SCC Local Configuration** dialog box. Choose the platform for which you wish to retrieve files. After you retrieve the files, go back and reset the platform to its original value.

A

References

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

LabVIEW Function and VI Reference Manual. A useful sample of quality documentation for libraries of VIs.

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, and G. Rymar. Proceedings from 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. Edited by Lynda P. Gruggett, LTR Publishing, phone (214) 706-0587, fax (214) 706-0506. A quarterly disk of VIs and a newsletter that features technical articles about all aspects of LabVIEW.

Rapid Development. Steve McConnell, Microsoft Press. Explanation of software engineering practices in a 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. Contains interesting discussions of what Microsoft has done right and what it has done wrong. Includes 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 bring problems and solutions into focus.

Software Engineering—A Practitioner's Approach. Roger S. Pressman, McGraw-Hill Inc., 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 wideband 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.

B

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 USA 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.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Québec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
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 6120092	03 6120095
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
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

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: _____

_____

_____

_____

_____

Professional G Developers Tools Reference Manual Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use 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.

National Instruments Products

Hardware revision _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

National Instruments software _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Other Products

Computer make and model _____

Microprocessor _____

Clock frequency or speed _____

Type of video board installed _____

Operating system version _____

Operating system mode _____

Programming language _____

Programming language version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

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 Tools Reference Manual*

Edition Date: May 1998

Part Number: 321393B-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 _____

_____

E-Mail Address _____

Phone (___) _____ Fax (___) _____

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

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Glossary

A

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 to verify the module behaves the way it is supposed to according to the requirements specification.

C

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 software professionals use.
CIN	*See* Code Interface Node.
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 occur only through the Source Code Control mechanism. It is also common to implement release configuration management to ensure 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.

P

prototype	A simple, quick implementation of a particular task 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 conditions	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

Software Engineering Institute (SEI)	A federally funded research and development center chartered to study software engineering technology. The SEI is located at Carnegie Mellon University and is sponsored by the Defense Advanced Research Projects Agency. Refer to `http://www.sei.cmu.edu`. for more information.
source lines of code	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 depends on the organization. For example, some organizations do not count blank lines and comment lines. Some count C lines, and some count only the final assembly language lines.
spiral model	A lifecycle model that emphasizes risk management through a series of iterations in which risks are identified, evaluated, and dealt with.
system testing	System testing begins after integration testing is complete. System testing assures that all the individual components function correctly together and constitute a product that meets the intended requirements. This stage often uncovers performance, resource usage, and other problems.

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. For example, white box testing is used to verify all the paths of execution of the module have been exercised.
wideband delphi estimation	Wideband delphi is a technique used by a group to estimate the amount of effort a particular project will take.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W