LabVIEW™

Development Guidelines

NATIONAL INSTRUMENTS™

April 2003 Edition
Part Number 321393D-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 61 2 9672 8846, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599,
Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949,
Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530,
China 86 21 6555 7838, Czech Republic 42 02 2423 5774, Denmark 45 45 76 26 00,
Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427,
Hong Kong 2645 3186, India 91 80 4190000, Israel 972 0 3 6393737, Italy 39 02 413091,
Japan 81 3 5472 2970, Korea 82 02 3451 3400, Malaysia 603 9059 6711, Mexico 001 800 010 0793,
Netherlands 31 0 348 433 466, New Zealand 64 09 914 0488, Norway 47 0 32 27 73 00,
Poland 48 0 22 3390 150, Portugal 351 210 311 210, Russia 7 095 238 7139, Singapore 65 6 226 5886,
Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00,
Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to techpubs@ni.com.

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 document 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

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or ni.com/patents.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Contents

Chapter 3
Prototyping and Design Techniques

Chapter 4
Scheduling and Project Tracking

Chapter 5
Creating Documentation

Chapter 6
LabVIEW Style Guide

Appendix A
References

Appendix B
Technical Support and Professional Services

Glossary

Index

Figures

Table

About This Manual

The LabVIEW Development Guidelines describe many of the issues that arise when developing large applications. The guidelines are based on the advice of LabVIEW developers and provide a basic survey of software engineering techniques you might find useful when developing your own projects.

This manual also includes a discussion of style for creating VIs. Developers who have used LabVIEW and are comfortable in the LabVIEW environment can use the LabVIEW Development Guidelines to maintain a consistent style in their projects.

Conventions

The following conventions appear in this manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.
	This icon denotes a tip, which alerts you to advisory information.
bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should 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 code excerpts.

Related Documentation

The following documents contain information that you might find helpful as you read this manual:

- *LabVIEW Help*, available by selecting **Help»VI, Function, & How-To Help**
- *LabVIEW User Manual*
- *LabVIEW Custom Controls, Indicators, and Type Definitions* Application Note
- *LabVIEW Performance and Memory Management* Application Note
- *LabVIEW Unit Validation Test Procedure* Application Note
- *Porting and Localizing LabVIEW VIs* Application Note
- *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability* Application Note

1 Development Models

This chapter provides examples of common development problems and describes different software engineering lifecycle models.

LabVIEW makes it easy to assemble components of data acquisition, test, and control systems. Because creating applications in LabVIEW is so easy, many people begin to develop VIs immediately with relatively little planning. For simple applications, such as quick lab tests or monitoring applications, this approach can be appropriate. However, for larger development projects, good project planning is vital.

Common Development Pitfalls

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

- "I have not really thought it through, but I think that we can complete the project in…."
- Estimates made without planning rarely are accurate because they usually are based on an incomplete understanding of the problem. When developing for someone else, often you each have different ideas about the project requirements. To make accurate estimates, you both must understand the requirements and work through a preliminary high-level design to understand the components you need to develop.
- "I think I understand the problem the customer wants to solve, so I can begin development immediately."

  There are two problems with this statement. The first problem is that a lack of consensus on project goals results in schedule delays. Your idea of what a customer wants might be based on inadequate communication. Refer to the *Lifecycle Models* section of this chapter for more information about developing a list of requirements and prototyping a system. Requirement lists and prototypes are two useful tools for clarifying project goals.

A second problem with this statement is that immediately beginning development frequently means writing code without a detailed design. Just as builders do not construct a building without architectural plans, developers do not begin building an application without a detailed design. Refer to the *Code and Fix Model* section of this chapter for more information about different development models you can follow.

- "We do not have time to write detailed plans. We have a strict schedule, so we need to begin developing immediately."

  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 steps because planning does not seem as productive as developing code. As a result, developers create VIs without a clear idea of how the VIs all work together. Consequently, you have to rework code 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 of this chapter and Chapter 3, *Prototyping and Design Techniques*, for more information about better approaches to developing software.

- "We can incorporate all the features in the first release. If the application does not include every feature, it is useless."

  In some cases, this statement is correct. However, for most applications, developing in stages is a better approach. When you analyze the requirements for a project, prioritize the features. You can develop an initial VI 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 a timely software release. Refer to the *Lifecycle Models* section of this chapter for more information about using development models.

- "If I can just get all the features in within the next month, I can fix any problems before the software releases."

  To release high-quality products on time, you must maintain quality standards throughout development. Do not build new features on an unstable foundation and rely on correcting problems later. This development technique exacerbates problems and increases cost. Even if you complete all the features on time, the time required to correct the problems in the existing code and the new code can delay the release of the product. 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 2, *Incorporating Quality into the*

Development Process, for more information about techniques for producing high-quality software.

- "We are behind in the project schedule. We need to increase the number of developers working on the project."

  In many cases, increasing the number of developers actually delays the project. Adding developers to a project requires time for training, which takes away time originally scheduled for development. Add resources earlier in the project rather than later. There also is a limit to the number of people who can work on a project effectively. With fewer people, there is less overlap. You can partition the project so each person works on a particular section. The more developers you add, the more difficult it becomes to avoid overlap. Refer to Chapter 2, *Incorporating Quality into the Development Process*, for more information about configuration management techniques that can help minimize overlap. Refer to Chapter 3, *Prototyping and Design Techniques*, for more information about methods for partitioning software for multiple developers.

- "We are behind in the 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 adjust your schedule. Assuming you can make up lost time can postpone choices until it becomes too costly to deal with them. For example, if you realize in the first month of a six-month project that you are behind, you can sacrifice a planned feature or add time to the overall schedule. If you do not realize you are behind schedule until the fifth month, you can affect decisions that other developers on your team made. Changing these decisions can be costly.

  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.

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 they must solve.
- Have a flexible development strategy that minimizes risk and accommodates change.

Lifecycle Models

Software development projects are complex. To deal with these complexities, many developers adhere to 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 steps to follow when developing software—from the initial concept stage to the release, maintenance, and subsequent upgrading of the software.

Many different lifecycle models currently exist. 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 you can customize these models to fit the requirements of a project.

Although this section is theoretical in its discussion, in practice consider all the steps these models encompass. Consider how you decide what requirements and specifications the project must meet and how you deal with changes to them. Also consider when you need to meet these requirements and what happens if you do not meet a deadline.

The lifecycle model is a foundation for the entire development process. Good decisions 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 they occur, 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 usually have to rewrite large parts of the application. Alternative development models can help you catch these problems in the early concept stages, when making changes is easier and less expensive.

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. This model is one of the oldest models and is widely used in government projects and in many major companies. Because the model emphasizes planning in the early stages, it catches design flaws before they develop. Also, because the model is document and planning intensive, it works well for projects in which quality control is a major concern.

The pure waterfall lifecycle consists of several non-overlapping stages, as shown in Figure 1-1. The model begins with establishing system requirements and software requirements and continues with architectural design, detailed design, coding, testing, and maintenance. The waterfall model serves as a baseline for many other lifecycle models.

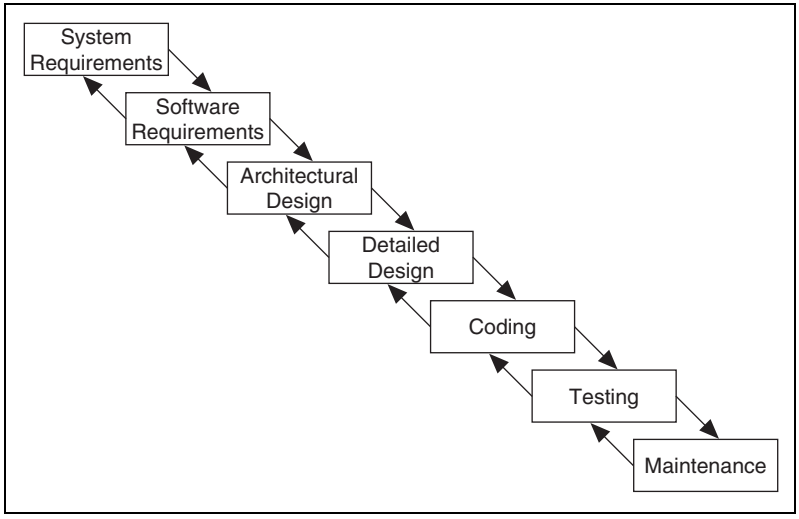


Figure 1-1. Waterfall Lifecycle Model

- System requirements—Establishes the components for building the system, including the hardware requirements, software tools, and other necessary components. Examples include decisions on hardware, such as plug-in boards (number of channels, acquisition speed, and so on), and decisions on external pieces of software, such as databases or libraries.
- Software requirements—Establishes the expectations for software functionality and identifies which system requirements the software affects. Requirements analysis includes 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 the interaction of those components, but the design does not define the structure of each component. You also determine the external interfaces and tools to use in the project.
- Detailed design—Examines the software components defined in the architectural design stage. 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—Addresses problems and enhancement requests after the software releases.

  In some organizations, a change control board maintains the quality of the product by reviewing each change made in the maintenance stage. Consider applying the full waterfall development cycle model when correcting problems or implementing these enhancement requests.

In each stage, you create documents that explain the 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. You also can incorporate prototyping into any stage from the architectural design and after. Refer to the *Prototyping* section of this chapter for more information about using prototyping in projects.

Many people believe you cannot 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 can 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 delay 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, 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, thus reducing the documentation requirements and the cost of returning to earlier stages to revise them. Another common modification is to incorporate prototyping into the requirements phases. Refer to the *Prototyping* section of this chapter for more information about prototyping.

Overlapping stages, such as the requirements stage and the design stage, make it possible to integrate feedback from the design phase into the requirements. However, overlapping stages can make it difficult to know when you are finished with a given stage. Consequently, progress is more difficult to track. Without distinct stages, problems can cause you to defer important decisions until later 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 can discover that you need to adjust the requirements.

Prototyping is an effective tool for demonstrating how a design meets 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 the overall objectives. In addition to clarifying the requirements, a 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.

However, prototyping has its drawbacks. Because it appears that you have a working system, customers may expect a complete system sooner than is possible. In most cases, prototypes are built on compromises that allow it to come together quickly but prevent the prototype from being an effective basis for future development. You need to decide early if you want to use

the prototype as a basis for future development. All parties need to agree with this decision before development begins.

Be careful that prototyping does not become a disguise for a code and fix development cycle. Before you begin prototyping, gather clear requirements and create a design plan. Limit the amount of time you spend prototyping before you begin. Time limits help to avoid overdoing the prototyping phase. As you incorporate changes, update the requirements and the current design. After you finish prototyping, consider returning to one of the other development models. For example, consider prototyping as part of the requirements or design phases of the waterfall model.

LabVIEW Prototyping Methods

There are a number of ways to prototype a system in LabVIEW. In systems with I/O requirements that are 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 use to display data or prompt the user for settings is difficult on paper. Instead, consider designing VI front panels with the controls and indicators you need. Leave the block diagram empty and figure out how the controls work and how various actions require other front panels. For more extensive prototypes, tie the front panels together. However, do not get carried away with this process.

If you are bidding on a project for a client, using front panel prototypes is an extremely effective way to discuss with the client how you can satisfy his or her requirements. Because you can add and remove controls quickly, especially if the block diagrams are empty, you help customers clarify 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 1-2 illustrates the spiral lifecycle model.

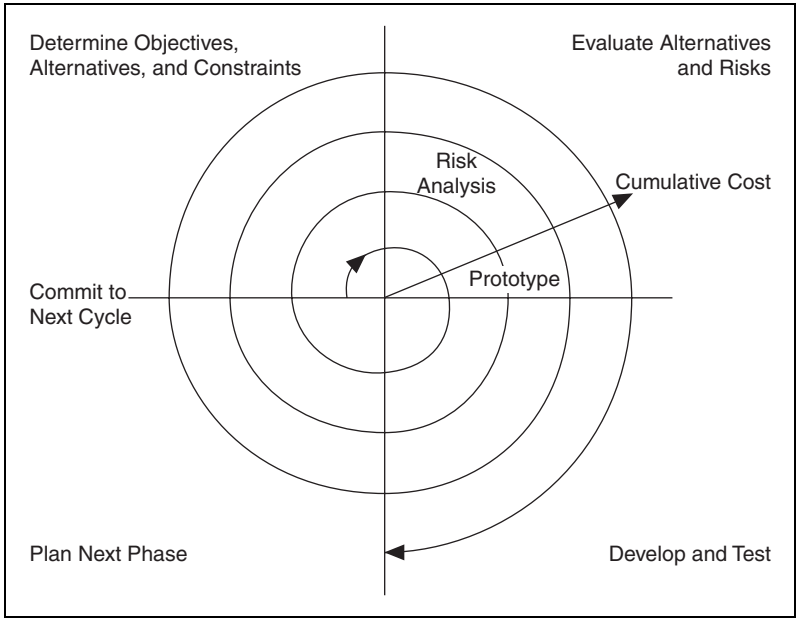


Figure 1-2. Spiral Lifecycle Model

Risks are any issues that are not clearly defined or have the potential to affect the project adversely. For each risk, consider the following two things:

- The likelihood of the risk occurring (probability)
- The severity of the effect of the risk on the project (loss)

You can use a scale of 1 to 10 for each of these items, with 1 representing the lowest probability or loss and 10 representing the highest. Risk exposure is the product of these two rankings.

Use a table such as the one in Table 1-1 to keep track of the top risk items of the project.

Table 1-1. 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, deal with the risks with the highest risk exposure first. In this example, the first spiral deals with the potential of the data acquisition rates being too high. If after the first spiral, you demonstrate that the rates are high, you can change to a different hardware configuration to meet the acquisition requirements. Each iteration can identify new risks. In this example, using more powerful hardware can introduce higher costs as a new 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, 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. It is possible that none of the options are viable because the options are too expensive, time-consuming, or do not meet the requirements.

The advantage of the spiral model over the waterfall model is that you can evaluate which risks to handle 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, assumptions about the risky components can spread throughout the design, and when you discover the problems, the rework involved can be very expensive.

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 can 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 can apply a waterfall lifecycle model to the design, coding, testing, and maintenance stages.

Other lifecycle models exist. Refer to Appendix A, *References*, for a list of resources that contain information about other development methodologies.

2

Incorporating Quality into the Development Process

This chapter describes strategies for producing quality software.

Many developers who follow the code and fix programming style described in Chapter 1, *Development Models*, mistakenly believe quality is not an issue until the testing phase. Developers must design quality into a product from the start. Developing quality software begins by selecting a development model that helps you avoid problems from the beginning. Consider quality during all stages of development—requirements and specifications, design, coding, testing, release, and maintenance.

Do not regard quality controls as tedious requirements that impede development. Most quality controls streamline development so you discover the problems before they are in the software, when it is inexpensive to fix them.

Quality Requirements

Set the quality standards for a product during the requirements stage. Treat the desired quality level as a requirement. Weigh the merits and costs of various options you have for applying quality measures to the project. Some of the trade-offs to consider include ease of use versus power and complexity, and speed versus robustness.

For short projects, 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 must be capable of handling invalid inputs. For example, if an operator mistakenly selects invalid voltage or current settings, the application must handle them 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 allows 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 1, *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 that occur later in the process need to be implemented. For example, customers often introduce new requirements in the design stage, or performance problems discovered during development prompt a reevaluation of the design. You also may need to rewrite a section of code to correct a problem found in testing. Changes can affect any component of the project from the requirements and specification to the design, code, and tests. If these changes are not made carefully, you can introduce new 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 process is important for projects with multiple developers. As the developers work on VIs, they need a method for collecting and sharing work. A simple method to deal with this is to establish a central source repository. If all the computers of developers are on the network, 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 can easily merge changes into the project. The other developer must use the Compare VIs tool, available in the LabVIEW Professional Development System, to determine the differences and merge the changes into a new version. Avoid this situation by ensuring good communication among the developers. If each developer notifies the others when he needs to work on a specific VI, the others know not to work

on that VI. However, remain flexible because inevitably, mistakes occur and work is lost.

Source code control tools are a solution to the problem 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 create a project, you can check out a file for development. Checking out a file marks it with your name so no other developer can modify the file. However, other developers can retrieve the current version of the file from the server. To modify a file, check out the file, make changes, test the changes, and check the file back into the source code control system. After checking in the file, the latest version of the file is available to the whole development team again. Another developer can then check out the file to make further modifications.

Managing Project-Related Files

Source code control tools can manage more than just VIs. Use these tools to manage all aspects of a project—requirements, specifications, illustrations, reviews, and other documents related to the project. This method ensures that you control access to these documents and share them as needed. Also use these tools to track changes to files and access older versions of files.

Chapter 3, *Prototyping and Design Techniques*, describes how source management of all project-related files is extremely important for developing quality software. 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. For example, you might change a file, check it in, and then you realize you made a mistake. Another reason for retrieving an old version of a file or project is to send a beta version of the software to a customer and continue development. If the customer reports a problem, you can access a copy of the beta version of the software.

One way to access an old version of a file or project is to back up files periodically. However, unless you back up the VI after every change, you do not have access to every version.

Source code control tools provide a way to check in new versions of a file and make a backup 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 source code control tools to label versions of files with descriptive names like `beta`, `v1.0`. 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 the software, label the files specifically for that version.

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 many changes.

Source code control tools maintain a log of all changes made to files and projects. When checking in a file, the tools prompt the developer to write a summary of the changes made. The tool adds this summary information to the log for that file.

You can view the history information for a file or for the system and generate reports that contain the summary information.

In addition, if you back up the project at specific checkpoints, you can use the Compare VIs tool to compare the latest version of a project with another version to verify the changes in the project.

Change Control

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

Cautiously deal with changes to specific components, such as documents related to user requirements, 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 can be the customer, other departments in your company can be the customer, or you can develop the software under contract for a third party. When you are the customer, adjusting requirements as you move through the specification and even the design stage is much easier. If you are developing for someone else, changing requirements is more difficult.

Source code control tools give you a degree of control when making changes. You can track 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, Rational Software ClearCase, or Perforce, 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, consider limiting change privileges for requirement documents to specific team members. You also can control access so a user has privileges to modify a file only with the approval of the change request.

The amount of control you apply varies throughout the development process. In the early stages of the project, before formal evaluation of the requirements, you do not need to strictly 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. Apply the same concept of varying the level of control to specifications, test plans, and code before and after completing a project phase.

Testing Guidelines

Decide up front what level of testing the project requires. Engineers under deadline pressure frequently give less attention to testing, devoting more time to other development. With a certain level of testing, you are guaranteed to save time.

Developers must clearly understand the degree to which you expect testing. Also, testing methodologies must be standardized, and results of tests must be tracked. As you develop the requirements and design specifications, also develop a test plan to help you verify that the system and all its components work. Testing reflects the quality goals you want to achieve. For example, if performance is more critical than robustness, develop more tests for performance and fewer that attempt incorrect input or low-memory situations.

Testing is not an afterthought. Consider testing as part of the initial design phases and test 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 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. You can perform black box testing based largely on knowledge of the requirements and the interface of a module. For a subVI, you can perform black box tests on the interface of a subVI to evaluate results for various input values. If robustness is a quality goal, include erroneous input data to see if the subVI successfully deals with it. For example, for numeric inputs, see how the subVI deals with Infinity, `<Not A Number>`, and other out-of-range values. Refer to the *Unit Testing* section of 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 work. 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 testing all possible paths is difficult.

Although white box testing is difficult to fully implement for large programs, you can choose to test the most important or complex paths. You can combine white box testing with black box testing for more thorough software testing.

Unit, Integration, and System Testing

Use black box and white box testing 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 the 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 can 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 on the block diagram are executed and performed correctly. Individual developers can perform unit tests as they work on the 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 path names.
- What happens when the user clicks the **Cancel** button in a file dialog box?
- What happens if the user aborts the VI?

Define various sets of inputs that thoroughly test the VI, write a test VI that calls the VI with each combination of inputs, and checks the results. Use interactive datalogging to create input sets, or test vectors, and retrieve them interactively to re-test the VI. You also can automatically retrieve data from a test VI that performs datalogging programmatically. Refer to the *LabVIEW Unit Validation Test Procedure* Application Note for more information about testing VIs.

To perform unit testing, you may need to create a stub of some components that you have not implemented yet or that you are developing still. 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 the components early, you can create a stub of the file I/O driver by creating a VI with the same interface. This stub 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. Refer to the *Integration Testing* section for more information about testing during the integration phase.

Regardless of how you test VIs, record exactly how, when, and what you tested and keep any test VIs you created. This test documentation is especially important if you create VIs for paying customers and for internal use. When you revise the VIs, run the existing tests to ensure you have not broken anything. Also update the tests for any new functionality you added.

Refer to the *LabVIEW Unit Validation Test Procedure* Application Note for more information about unit testing.

Integration Testing

You perform integration testing on a combination of units. Unit testing usually finds most bugs, but integration testing can reveal unanticipated problems. Modules might not work together as expected. They can interact in unexpected ways because of the way they manipulate shared data. Refer to the *LabVIEW Performance and Memory Management* Application Note for more information about possible problems that are discovered during testing.

You also can perform integration testing 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 can develop unit tests to verify that each subVI correctly sends the appropriate commands. He also can develop integration tests that use several of the subVIs in conjunction with each other to verify that there are no unexpected interactions.

Do not perform integration testing as a comprehensive test in which you combine all the components and try to test the top-level program. This method can be expensive because it is difficult to determine the specific source of problems within a large set of VIs. Instead, 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 implemented as stubs, as described in the *Unit Testing* section of this chapter. Once you verify 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 of this chapter. After you combine a set of modules and verify that they work together, add components and test them with the already-debugged subsystem.

The bottom-up approach consists of tests that gradually increase in 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 previously tested features still work. Regression testing consists of repeating some or all previous tests. If you need to perform the same tests numerous times, consider developing representative subsets of tests to use for frequent regression tests. You can

run these subsets of tests at each stage. You can run the more detailed tests to test an individual set of modules if problems arise or as part of a more detailed regression test that periodically occurs during development.

System Testing

System testing happens after integration to determine if the product meets customer expectations and to make sure the software works as expected within the hardware system. You can test the system using a set of black box tests to verify that you have met the requirements. With system testing, you test the software to make sure it fits into the overall system as expected. Most LabVIEW applications perform some kind of I/O and also communicate with other applications. Make sure you test how the application communicates with other applications.

When testing the system, consider the following questions:

- 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 developers did not consider or complete. With alpha testing, test a functionally complete product in-house to see if any problems arise. With beta testing, when alpha testing is complete, the customers in the field test the product.

Alpha and beta testing are the only testing mechanisms for some companies. Unfortunately, alpha and beta testing actually can be inexact and are not a substitute for other forms of testing that rigorously test each component to verify that the component meets stated objectives. When 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 testing methodologies attempt to find problems by exploration. But many software engineers are proponents of formal verification of software. Formal methods of verification 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. You can perform this process 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. Refer to Appendix A, *References*, for information about documents that include more information about the verification process.

This type of testing becomes more complex as 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 verification testing 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 own style of development, color preferences, display techniques, documentation practices, and block diagram methodologies. One developer may use global variables and sequence structures extensively, while another may prefer to use data flow.

Inconsistent style techniques create software that looks bad. If VIs have different behaviors for the same button, users can become confused and find the user interface of the VIs difficult to use. For example, if one VI requires a user to click a button when he completes a task and another VI requires the user to use a keyboard function key to signal completion of the task, the user can become confused as to what he needs to do.

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 is difficult to modify.

Establish an initial set of guidelines for your VI development team and add additional rules as the project progresses. You can use these style guidelines in future projects.

Chapter 6, *LabVIEW Style Guide*, provides some style recommendations and checklists to use when working on LabVIEW projects. Use these recommendations and checklists as a basis for developing your own style guidelines. To create a style guide customized to fit the specifications of your project, add, remove, or modify the guidelines provided.

Creating a single standard for development style in any programming language is very difficult because each developer has his own development

style. Select a set of guidelines that works for you and your development team and make sure everyone follows those guidelines.

Design Reviews

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

- 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 for features that are essential as opposed to features that are extras. Extra features are okay, but if quality and schedule are important, schedule these extra features for later in the development process. This way you can drop or move the list of extra features to subsequent releases if time becomes a constraining factor. Document the results of the design review and any recommended changes.

Code Reviews

A code review 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 also can perform the review online because VIs are easier to read and navigate online. Talk through the design and compare the description to the actual implementation. Consider many of the same issues included in a design review. During a code review, ask and answer some of the following questions:

- 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 that a parallel block diagram simultaneously attempts 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? Refer to the *LabVIEW Performance and Memory Management* Application Note for more information about creating effective code.

- Is the block diagram easy to maintain? Does the developer make good use of hierarchy, or is he 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 review. Take notes on the problems you encounter and add them to a list you can use as a guideline for other walk-throughs.

Focus on technical issues when doing a code review. Remember to review only the code, not the developer who produced it. Do not focus only on the negative; be sure to raise positive points as well.

Refer to Appendix A, *References*, for a list of resources that contain more information about walk-through techniques.

Post-Project Analysis

At the end of each stage in the development process, consider having a post-project analysis meeting to discuss what went well and what did not. Each developer must evaluate the project honestly and discuss obstacles that reduced the quality level of the project. Consider the following questions during a post-project analysis meeting:

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

Post-project analysis meetings at major milestones can help to correct problems mid-schedule instead of waiting until the end of the release cycle.

Software Quality Standards

As software becomes a more critical component in systems, concerns about software quality are increasing. 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, consider following one of these standards.

The following sections include a brief overview of the most recognized standards. Refer to Appendix A, *References*, for a list of resources that contain more information about these standards.

International Organization for Standardization ISO 9000

The International Organization for Standardization (ISO) developed the ISO 9000 family of standards for quality management and assurance. Many countries adopted these standards. In some cases, government agencies require compliance with this ISO standard. A third-party auditor generally certifies compliance. The ISO 9000 family of standards is used widely within Europe and Asia. It has not been widely adopted within the United States, although many companies and some government agencies are beginning to use it.

Each country refers to the ISO 9000 family of standards by slightly different names. For example, the United States adopted the ISO 9000 as the ANSI/American Society for Quality Control (ASQC) Q90 Series. In Europe, the European Committee for Standardization (CEN) and the European Committee for Electrotechnical Standardization (CENELEC) adopted the ISO 9000 as the European Norm (EN) 29000 Series. In Canada, the Canadian Standards Association (CSA) adopted the ISO 9000 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 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 is 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 documented practices 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 Web site at `www.fda.gov` for more information about the 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 the processes of an organization.

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 were successful. Groups use configuration management tools to manage projects. They also 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 are not usually 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 2-1 illustrates the five levels of the CMM and the processes necessary for advancement to the next level.

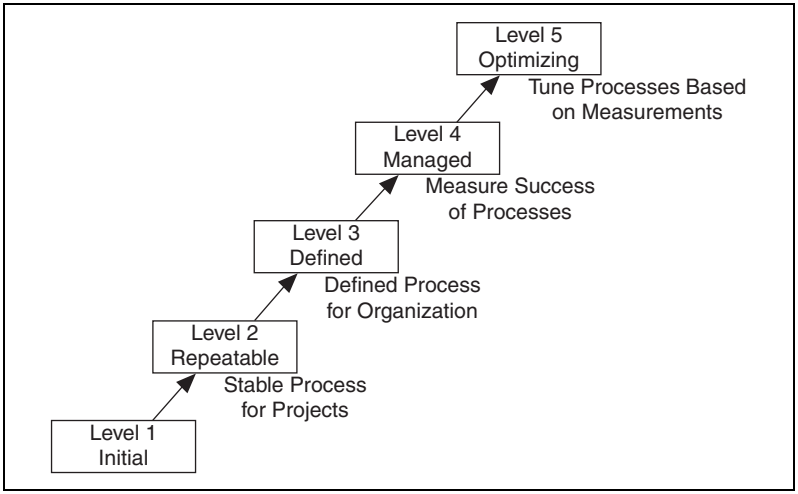


Figure 2-1. Capability Maturity Model

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. The ISO specifications are fairly high-level documents, consisting of only a few pages. The CMM is a very detailed document, consisting of 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 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 may 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.

3

Prototyping and Design Techniques

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

When you first begin a programming project, deciding how to start can be intimidating. Many LabVIEW developers start immediately with a code and fix development process, building some of the VIs they think are needed. Then they realize they actually need something different from what they have built already. Consequently, people unnecessarily develop, rework, or discard code.

Defining the Requirements of the Application

Before you develop a detailed design of a system, define the goals clearly. 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. Do some research at this early stage to be sure you can meet the specifications. Other requirements depend on user preferences, such as file formats or graph styles.

Try to distinguish between absolute requirements and desires. While you can try to satisfy all requests, it is best to have an idea about which features you can sacrifice if you run out of time.

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. However, cost also is a constraint. Include all these issues in the requirements. If you can avoid specifying the hardware, you can adjust the design after you begin prototyping and benchmarking various components. As long as the costs are within the specified budget and the timing and precision issues are met, the customer may 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 can be helpful in explaining 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 probably is a simpler solution that produces the same results at a lower cost in a shorter time period.

Top-Down Design

The block diagram programming metaphor LabVIEW uses is designed to be easy to understand. Most engineers already use block diagrams to describe systems. The block diagram makes it easier to convert 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 the VI. For example, you can 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 displaying the results, a block for saving the data to disk, and a block to clean up at the end of the VI.

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 the 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 a subsequent VI needs. If so, make sure the top-level block diagram sketch contains wires to pass the data between the VIs. Select **File»VI Properties** and select **Documentation** from the **Category** pull-down menu to document the functionality of the VI and the inputs and outputs.

In analyzing the transfer of data from one block to another, avoid global variables because they hide the data dependency among VIs and can introduce race conditions. Refer to the *LabVIEW Performance and Memory Management* Application Note for more information about issues that can arise when creating VIs. As the VI becomes larger, it becomes

difficult to debug if you use global variables as the method of transferring information among VIs.

Continue to refine the design by breaking down each of the component blocks into more detailed outlines. For example, you can fill out the block diagram of what was once a stub VI and then place those lower level stub VIs that represent each of the major actions the VI must perform on the block diagram.

Be careful not to immediately begin implementing the system at this point. You must gradually refine the design so you can determine if you have left out any necessary components at higher levels. For example, when refining the acquisition phase, you may realize there is more information you need from the configuration phase. If you completely implement one block before you analyze a subsequent block, you may 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 the complexity.

Data Acquisition System Example

This example describes how you can apply top-down design techniques to a data acquisition system. This system must let the user provide some configuration of the acquisition, such as rates, channels, and so on. Then the system must acquire, process, and save the data to disk.

Start to design the VI hierarchy by breaking the problem into logical pieces. The flowchart in Figure 3-1 shows several major blocks you can expect to see in every data acquisition system.

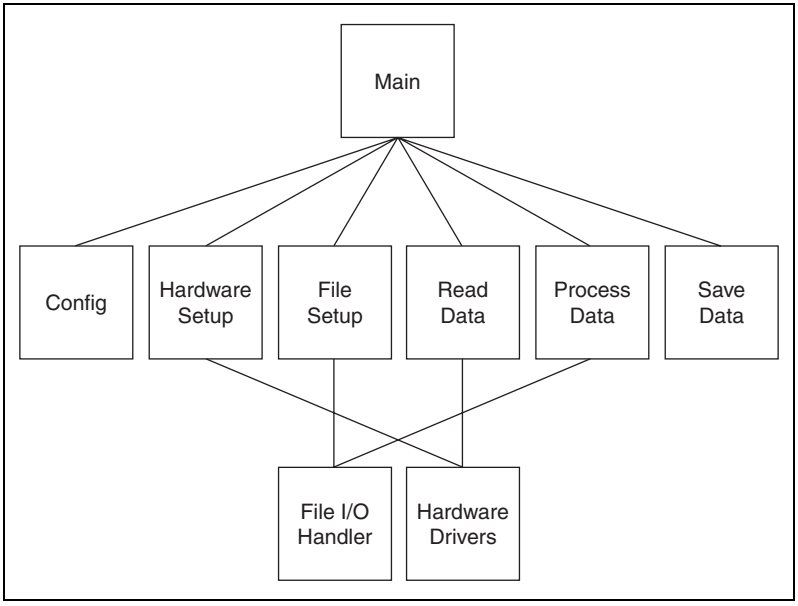


Figure 3-1. Flowchart of a Data Acquisition System

Think about the data structures needed, asking questions such as, "What information needs to accompany the raw data values from the Read Data VI to the Save Data VI?" The answer implies a cluster array, which is an array of many elements, of which each element 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. Modern structured programming reflects the usage of algorithms and data structures. Both work well in LabVIEW. This technique of programming also is suitable for pseudocode.
Figures 3-2 and 3-3 show a relationship between pseudocode and LabVIEW structures.

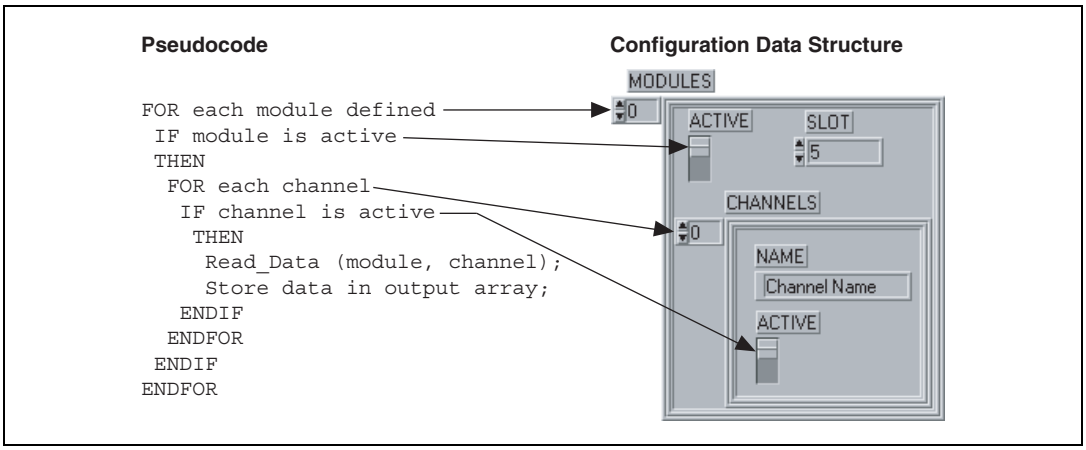


Figure 3-2. Mapping Pseudocode into a LabVIEW Data Structure

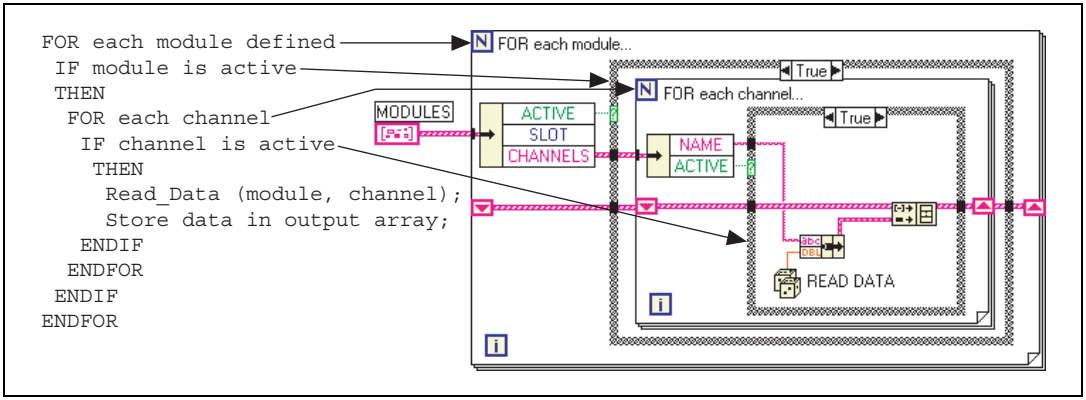


Figure 3-3. Mapping Pseudocode into Actual LabVIEW Code

Notice that the program and the data structure correspond in Figure 3-2.

Many experienced LabVIEW users prefer to use sketches of LabVIEW code. You can draw sketches of the familiar structures and wire them together on paper. This method is a good way to think things through, sometimes with the help of other LabVIEW programmers.

If you are not sure how a certain function will work, prototype it in a simple test VI, as shown in Figure 3-4. Artificial data dependency between the VIs and the main While Loop eliminates the need for a sequence structure.

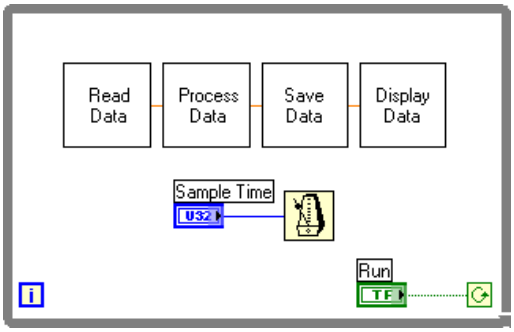


Figure 3-4. Data Flow for a Generic Data Acquisition Program

Finally, you are ready to write the program in LabVIEW. Remember to make the 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 the specific ones. Test the subVIs as you write them. Testing can involve constructing higher level test routines. However, catching the problems in one small module is easier than in a large hierarchy of VIs.

Bottom-Up Design

Usually, avoid bottom-up system design as a primary design method. When used in conjunction with top-down design, bottom-up design can be useful. If you start by building the lower level components and then progressing up the hierarchy, you gradually put 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 can build pieces that do not fit together correctly.

There are specific cases in which using bottom-up design is appropriate. If the design is constrained by low-level functionality, you can build that low-level functionality first to get an idea of how it will be used. This case is 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 can separate the design so configuring an instrument and reading a measurement from the instrument are done in distinct VIs. But if the instrument command set is more constraining than you thought, you may need to combine these operations. In this case, using a bottom-up strategy, you can start by building VIs that deal with the instrument command set.

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

The following example describes 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. For such an instrument, a bottom-up approach can be the most effective way to design a driver. 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 based on similar instrument drivers, to satisfy the needs of the user. Designing a driver requires more than putting knobs on GPIB commands. For example, the Tektronix 370A Curve Tracer has about 100 GPIB commands if you include the read and write versions of each command.

Once you begin programming, the hierarchy fills 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 example, 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. A communications handler also 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 perform a useful function. The driver VI quickly finds any fundamental flaws in earlier choices of data structures, terminal assignments, and default values.

Refer to the *LabVIEW Help* for more information about developing instrument drivers.

The top-level VI in Figure 3-5 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.

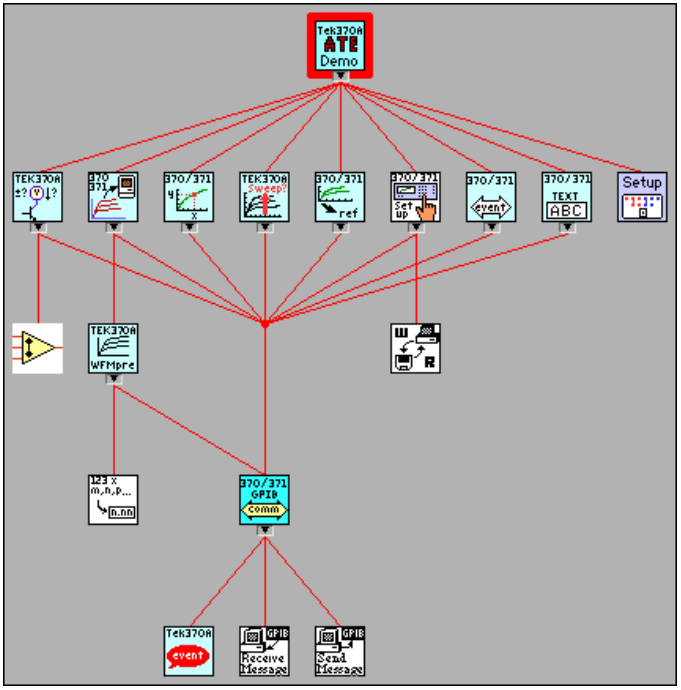


Figure 3-5. VI Hierarchy for the Tektronix 370A

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, look for areas that have a minimal amount of overlap. For example, a complicated data monitoring system can have one set of VIs to display and manipulate data and another set of VIs to acquire and save the information. These two modules are substantial, do not overlap, and can be assigned to different developers.

Inevitably, there is some interaction among 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 have access to the data it needs to display. The acquisition component needs to provide this information for the other module. At an early stage in development, consider designing the connector panes of VIs needed to transfer information between the two modules. Likewise, if there are global data structures that must be shared, analyze and define them early in the

architectural design stage, before the individual developers begin working on the components.

In the early stages, each developer can create stub VIs with the connector pane interface defined for a shared module. This stub VI can do nothing, or, if it is a VI that returns information, it can generate random data. This flexibility allows each member of the development team to continue development without having to wait for the completion of other modules. It also makes it easy for the individuals to perform unit testing of modules as described in Chapter 2, *Incorporating Quality into the Development Process*.

As components near completion, you can integrate the modules by replacing the stub components with the real counterparts. Then you can perform integration testing to verify the system works as a whole. Refer to the *Integration Testing* section of Chapter 2, *Incorporating Quality into the Development Process*, for more information about integration testing.

Front Panel Prototyping

As described in Chapter 1, *Development Models*, front panel prototypes provide insight into the organization of the program. Assuming the 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, just create the front panels. As you create buttons, listboxes, and rings, think about what needs to 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 helps to define the requirements for a project and gives you a better idea of its scope.

However, prototyping cannot solve all development problems. You have to be careful how you present the prototype to customers. Prototypes can give an 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 you will rework much of it in the development phase.

Another danger in prototyping is overdoing it. Consider setting strict time goals for the amount of time you 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, not with I/O constraints, data types, or algorithm issues in the design. Identifying front panel issues can 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 or 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 can prove that the design assumptions are incorrect.

For example, if you plan to use an instrument in the data acquisition system, build some simple tests that perform the type of I/O you plan to use. While the specifications can indicate that the instrument can handle the application you are creating, you may find that triggering, for example, takes longer than you expected. You also may find 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 the application can reveal these kinds of problems. Refer to the timing template example VI in the `examples/general/structs.llb` directory for an example of how to time a process. Because timing can fluctuate from one run to another for a variety of reasons, 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 include system interrupts, screen updates, user interaction, and initial buffer allocation.

Identifying Common Operations

As you design programs, you may find that certain operations are performed frequently. Depending on the situation, using subVIs or loops to repeat an action can simplify a VI.

For example, consider Figure 3-6, where three similar operations run independently.

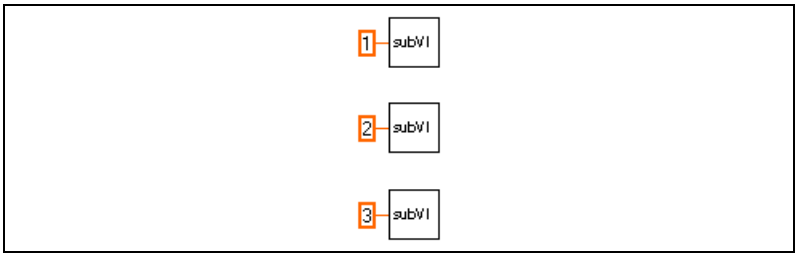


Figure 3-6. Operations Run Independently

An alternative to this design is a loop that performs the operation three times, as shown in Figure 3-7. You can build an array of the different arguments and use auto-indexing to set the correct value for each iteration of the For Loop.

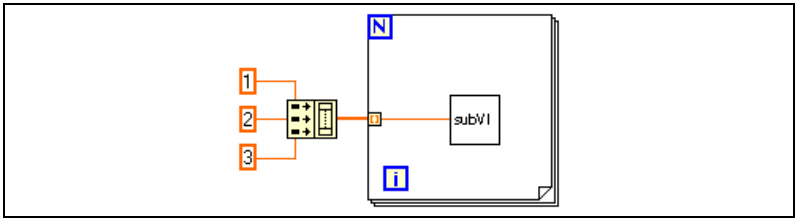


Figure 3-7. Loop Performs Operation Three Times

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 can add to the 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 minimized by any I/O you perform or by any memory management that occurs from complex manipulation of arrays.

4

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 eight hour work days. When creating schedules from estimates, consider dependencies such as the completion of a project before another can begin. Also consider daily tasks that take away from project time, such as meetings, support for existing projects, and so on.

Estimation

One goal of planning is to estimate the size of the project and then fit the project into the schedule because schedules partially, if not completely, drive most projects. Schedules, 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 sometimes have software projects exceed original estimates by a year or more. Poor planning or an incomplete idea of project goals often causes missed deadlines. Another major cause of missed deadlines is feature creep, which means the 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 measurable development model.

Poorly planned estimates are almost never accurate for the following reasons:

- People are usually overly optimistic. An estimate of two months at first seems like an infinite amount of time. However, during the last two weeks of a project, when developers find themselves working many overtime hours, it becomes clear that two months is too little time.
- The objectives, implementation issues, and quality requirements are not understood clearly. When challenged with the task of creating a data monitoring system, an engineer may estimate two weeks. If the

product is designed by the developer and for the developer, this estimate may be accurate. However, if the product is designed for other users, the developer probably is not considering requirements that a less knowledgeable user assumes but never clearly specifies.

For example, VIs need to be reliable and easy to use because the developer cannot be there to fix them if a problem occurs. Thus a considerable amount of testing and documentation is necessary. The user also needs to save results to disk, print reports, and view and manipulate the data on screen. If the user does not discuss the project in detail with the developer, the project can potentially fail.

- People often forget to account for daily tasks in their schedule. 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 unknown. The implementation of 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 more information about these and other estimation techniques.

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 partially because the information is easy to gather. Numerous programs exist for analyzing text-based 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, included in the LabVIEW Professional Development System, provides a method for measuring a corresponding metric for LabVIEW 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, including functions, VIs, and structures, such as loops and sequences, on the block diagram excluding labels and graphics. Refer to the *LabVIEW*

Help for more information about how to use this tool and the accounting mechanism it uses.

You can use the 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 need 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.

Using historical information, you then need to estimate the number of nodes required for a new project. You cannot do this for an entire project at once. Instead, divide the project into subprojects that you can compare to other tasks completed in the past. Once you have subprojects, estimate the number of nodes required for each subproject and produce a total estimate of nodes and the time required for development.

Problems with Source Lines of Code and Number of Nodes

Size-based metrics are not accepted uniformly in software engineering. Many people favor size-based metrics because they are relatively easy to gather and because there is a lot of literature on size-based metrics. Opponents of size-based metrics have the following arguments:

- Size-based metrics are dependent on the organization. Lines of code and numbers of nodes are useful in an organization if you are working with the same group of people who follow the same style guidelines. Using size-based metrics from other companies or groups can be difficult because of differing levels of experience, different expectations for testing and development methodologies, and so on.
- Size-based metrics also are dependent on the programming language. Comparing a line of code in assembly language to one written in C is like comparing apples to oranges. Statements in higher level programming languages can provide more functionality than those in lower level programming languages. Comparing numbers of nodes in LabVIEW to lines of code in a text-based programming language is 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. A block diagram that consists of 50 nested loops is much more difficult to understand than 50 subVIs connected together in a line.
- Size-based metrics rely on a solid base of information that associates productivity with various projects. To be accurate, have statistics for each member of a team because the experience level of team members varies.

Despite these problems, many developers use size-based metrics for estimating projects. A good technique is to estimate a project using size-based metrics in conjunction with one of the other methods described in this chapter. 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 are easier to estimate. A good guideline is to divide 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 takes to complete each task using uninterrupted eight hour work days as the unit of estimation. Have the developers list any assumptions made in forming estimates. Then reconvene the group to graph the overall estimates as a range of values. Keep the estimates anonymous and 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, the developers reexamine and adjust the estimates. The group then graphs and discusses the new estimates. This process can go on for three or four cycles.

In most cases, you 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 and ignore certain outlying values to create an estimate. If some tasks turn out to be too expensive for the time allowed, the developer can consider adding resources or scaling back the project.

Even if the estimate is incorrect, the discussion from the meetings provides a better 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.

Refer to Appendix A, *References*, for more information about the Wideband Delphi estimation method.

Other Estimation Techniques

Several other techniques exist for estimating development cost. The following list briefly describes some popular techniques:

- Function-Point Estimation—Function-point estimation differs considerably from the size-estimation techniques described earlier. Rather than dividing the project into tasks that are estimated separately, function points are based on a formula applied to a category breakdown of the project requirements. This method analyzes the requirements for features such as inputs, outputs, user inquiries, files, and external interfaces. These features are tallied and weighted. The results 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 work well primarily with database applications but also are applicable to other software areas. Function-point estimation is popular as a rough estimation method

because it is useful 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—COnstructive COst MOdel (COCOMO) 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.

Refer to Appendix A, *References*, for information about more documents that describe other estimation techniques in detail.

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. While you can accurately estimate that a VI takes only two weeks to develop, you still need to fit that development into the overall schedule. Remember that you will have meetings and other events during the development time. If you have other projects to complete first, or if you need to wait for another developer to complete his work before starting the project, then you must factor extra time into the schedule.

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

After estimating project time, try to develop a schedule that accounts for overhead estimates and project dependencies. Remember your other responsibilities and tasks, such as weekly meetings to attend, existing projects to support, and reports to write.

Record progress meeting time estimates and schedule estimates. Track project time and time spent on other tasks each week. This information probably varies from week to week, but determining an average is a useful reference for future scheduling. Recording this information helps you plan future projects accurately.

Tracking Schedules Using Milestones

Milestones are a crucial technique for measuring 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 the 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 already are complete.

Throughout development, strive to keep the quality level high. If you defer problems until you reach a milestone, you are deferring risks that can delay the schedule. Delaying problems makes it seem like you are making more progress than you actually are. It also 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. Determine minor milestones from the task list that you created as part of the estimation process.

Refer to Appendix A, *References*, for more information about creating major and minor milestones.

Responding to Missed Milestones

One of the biggest mistakes people make is missing a milestone and not reexamining the project as a consequence. After missing a milestone, many developers continue on the same schedule, assuming they can work harder and make up the time.

Instead, if you miss a milestone, evaluate the reasons you missed it. Is there a systematic problem that always affects 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 avoid accusations. Have the entire team make suggestions on how to get back on track. Solutions can include stopping development and returning to design for a period of time, deciding to cut back on certain features, adding no new features until all the problems are fixed, or renegotiating 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.

Do not be surprised if you miss a milestone. 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 two months of a year-long project, getting back on schedule is difficult.

5

Creating Documentation

By carefully documenting your VIs, you create VIs that are usable and maintainable. Thorough documentation reduces confusion and makes future modifications easier.

This chapter focuses on documentation that is specific to LabVIEW-based development. It does not address general documentation issues that apply to all software products.

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

- Design-related documentation—Includes requirements, specifications, detailed design plans, test plans, and change history documents.
- User documentation—Explains how to use the software.

The style of each of these documents is different. The audience for design-related documentation generally has extensive knowledge of the tools that you are documenting. The audience for user documentation has a lesser degree of understanding and experience with the software.

The size and style of each document can vary depending on the project. For simple tools that are used in-house, you probably do not need as much documentation. If you plan to sell the product, you must allow a significant amount of time to develop detailed user-oriented documentation to support the product. For products that must go through a quality certification process, such as a review by the U.S. Food and Drug Administration (FDA), you must ensure that the design-related documentation meets all the requirements.

Designing and Developing 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 the project. If the project must meet a quality standard such as the ISO 9000, the format and detail level of the

documentation is different from the format and detail level of an in-house project.

Refer to Appendix A, *References*, for a list of documents that contain more information about the types of documents to prepare as part of the development process.

LabVIEW includes features that simplify the process of creating documentation for the VIs you design.

- **History** window—Use the **History** window to record changes to a VI as you make them.
- **Print** dialog box—Use the **Print** dialog box to create printouts of the front panel, block diagram, connector pane, and description of a VI. You also can use it to print 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 text files, or generate HTML or RTF files that you can use to create compiled help files.

Developing User Documentation

Organizing the documentation systematically helps users learn about your product, VI, or application. Different users have different documentation needs. End users of VIs fall into the following two classes: end users of top-level VIs and end users of subVIs. Each of these users have different documentation needs. This section addresses techniques for creating and organizing documentation that helps both of these classes of users. The format of user documentation depends on the type of product you create.

Systematically Organizing Documentation

To make documentation more helpful for the user, consider organizing the documentation in a systematic way. You can divide the help information into three categories—concepts, procedures, and reference material. Create documentation that reflects these three categories.

For example, the shipping LabVIEW documentation set includes printed manuals, help files, and PDFs. The printed manuals contain conceptual information and the help files contain task-oriented procedures and VI/function reference information. Application notes delivered in PDFs contain information about advanced topics.

Documenting a Library of VIs

If the software you are creating is a library of VIs for use by other developers, such as an instrument driver or add-on package, create documents with a format similar to the *LabVIEW Help*. You can assume the audience has a working knowledge of LabVIEW because the audience is other developers. You can create documentation that contains 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 can include information such as the VI name and description, a picture of the connector pane, and a picture and description of the data type for each control and indicator on the connector pane.

To generate most of the documentation for VIs and controls, select **File» VI Properties** and select **Documentation** from the **Category** pull-down menu. Refer to the *Describing VIs, Controls, and Indicators* section of this chapter for more information about documenting VIs and controls.

Select **File»Print** to print VI documentation in a format almost identical to the format used in the VI and function reference information in the *LabVIEW Help*. Use the **Print** dialog box to save the documentation to a file and to create documentation for multiple VIs at once.

Documenting an Application

If you are developing an application for users who are unfamiliar with LabVIEW, the documentation requires more introductory material. You can create a document that provides system requirements, basic installation instructions, and an overview about how the package works. If the package uses I/O, include hardware requirements and any configuration instructions the user must complete before using the application.

For each front panel with which the user interacts, 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 front panel the user sees first documented first. You also can use the **Print** dialog box to create this documentation.

Creating Help Files

You can create online help or reference documents if you have the right development tools. Online help documents are formatted text documents that you can create using a word-processing program, such as Microsoft Word and help compiling tools. You also can create online help documents in HTML with an HTML help compiler. Use hidden text to create special help features such as links and hotspots.

Use the **Print** dialog box to help you create the source material for the help documents.

After creating the source documents, use a help compiler to create the help document. If you need help files on multiple platforms, use a help compiler for the specific platform for which you want to generate help files. After creating and compiling the help files, add them to the **Help** menu of LabVIEW or to your custom application by placing them in the `help` directory. Refer to Chapter 15, *Documenting and Printing VIs*, of the *LabVIEW User Manual* for more information about creating help files.

You also can link to the help files directly from a VI. Link your VI to the **Help** menu using the **VI Properties»Documentation** dialog box. Refer to the *LabVIEW Help* for more information about linking to the **Help** menu from VIs. You also can use the Help functions to link to topics in specific help files programmatically.

Describing VIs, Controls, and Indicators

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.

Creating VI Descriptions

Create, edit, and view VI descriptions by selecting **File»VI Properties** and selecting **Documentation** from the **Category** pull-down menu. The VI description is often the only source of information about a VI available to the user. The VI description appears in the **Context Help** window when you move the cursor over the VI icon and in any VI documentation you generate.

Include the following items in a VI description:

- An overview of the VI
- Instructions about how to use the VI
- Descriptions of the inputs and outputs

Refer to the *LabVIEW Help* for more information about creating and editing VI descriptions.

Documenting Front Panels

One way of providing important instructions is to place a block of text prominently on the front panel. A list of important steps is valuable. You can include instructions such as, "Select **File»VI Properties** for instructions" or "Select **Help»Show Help**." For long instructions, you can use a scrolling string control instead of a free label. When you finish entering the text, right-click the control and select **Data Operations»Make Current Value Default** from the shortcut menu to save the text.

If a text block takes up too much space on the front panel, use a **Help** button on the front panel instead. Include the instruction string in the help window that appears when the user clicks the **Help** button. Use the **Window Appearance** page in the **VI Properties** dialog box to configure this help window 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.

You also can use a **Help** button to open an entry in an online help file. Use the Help functions to open the **Context Help** window or to open a help file and link to a specific topic.

Creating Control and Indicator Descriptions

Include a description for every control and indicator. To create, edit, and view object descriptions, right-click the object and select **Description and Tip** from the shortcut menu. The object description appears in the **Context Help** window when you move the cursor over the object and in any VI documentation you generate.

Unless every object has a description, the user looking at a new VI has no choice but to guess the function of each control and indicator. Remember to enter a description when you create the object. If you copy the object to another VI, you also copy the description.

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

- Functionality
- Data type
- Valid range (for inputs)
- Default value (for inputs)

  You also can list the default value in parentheses as part of the control or indicator name.
- Behavior for special values (0, empty array, empty string, and so on)
- Additional information, such as if the user must set this value always, often, or rarely

Designate which inputs and outputs are required, recommended, and optional to prevent users from forgetting to wire subVI connections. To indicate the required, recommended, and optional inputs and outputs, right-click the connector pane and select **This Connection is»Required**, **Recommended**, or **Optional**.

6

LabVIEW Style Guide

This chapter describes recommended practices for good programming technique and style. Remember that these are only recommendations, not strict rules.

Chapter 2, *Incorporating Quality into the Development Process*, discusses how inconsistent style causes problems when multiple developers work on the same project. The resulting VIs are difficult to maintain and can confuse users. To avoid these problems, establish a set of style guidelines for VI development. You can establish an initial set of guidelines at the beginning of the project and add more guidelines as the project progresses. The most important thing is not the specific style you use, but the consistent use of that style.

This chapter includes style checklists to help you maintain quality as you develop VIs. To save time, review the list before and during development. To create a successful VI, consider who needs to use the VI and for what reasons. Remember the following:

- Users need a clear front panel.
- Developers need an easy to read block diagram.
- Users and developers need thorough documentation.

Use this chapter and the style checklists as a starting point for creating your own style guide. Customize this chapter to fit your development preferences and to meet the specifications of your project.

Organizing VIs in Directories

Organize the VIs in the file system to reflect the hierarchical nature of the software. Make 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 6-1. Save the main VIs in this directory and the subVIs in a subdirectory. If the subVIs have subVIs, continue the directory hierarchy downward.

When you create the directory, organize the VIs and subVIs modularly. Figure 6-1 shows a folder, `MyApp`, containing a VI-based application. The main VI, `MyApp.vi`, is in this folder along with the folders containing all the subVIs. Notice that these folders are organized according to the functionality of the subVIs.

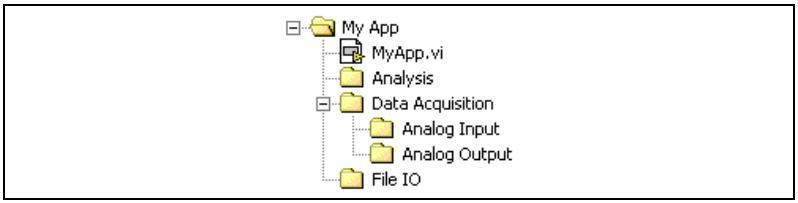


Figure 6-1. Directory Hierarchy

When naming VIs, VI libraries, and directories, avoid using characters that are not accepted by all file systems, such as slash (/), backslash (\), colon (:), and tilde (~). Most operating systems accept long descriptive names for files, up to 31 characters on Mac OS 9.*x* or earlier and 255 characters on other platforms.

Avoid creating files with the same name anywhere within the 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 links to the VI in memory. If you make backup copies of files, be sure to save them into a directory outside the normal search hierarchy so that LabVIEW does not mistakenly load them into memory when you open development VIs.

Refer to Chapter 7, *Creating VIs and SubVIs*, of the *LabVIEW User Manual* for more information about saving VIs individually and in VI libraries. Refer to the *LabVIEW Help* for more information about adding a new directory to the VI Search Path.

Front Panel Style

Front panels must be well-organized and easy to use because users see the front panel first when working with a VI. When designing a front panel, keep in mind two types of users, the end user and the developer. End users

work with user interface VIs, which have front panels that only the end user sees, and developers work with subVIs, which have front panels that only developers see.

Fonts and Text Characteristics

Do not use all the fonts and styles available. Limit your VI to the three standard fonts—application, system, and dialog—unless you have a specific reason to use a different font. For example, monospace fonts, which are fonts that are proportionally spaced, are useful for string controls and indicators where the number of characters is critical. Refer to the *LabVIEW Help* for more information about setting the default font, using custom fonts, and changing the font style.

The actual font used for the three standard fonts varies depending on the platform. For example, when working on Windows, preferences and video driver settings affect the size of the fonts. Text might appear larger or smaller on different systems, depending on these factors. To compensate for this, allow extra space for larger fonts and enable the **Size to Text** option on the shortcut menu. Use carriage returns to make multiline text instead of resizing the text frame.

To prevent labels from overlapping objects because of font changes on multiple platforms, allow extra space between controls. For example, if a label is to the left of an object, justify the label to the left and leave some space to the right of the text. If you center a label over or under an object, center the text of that label as well. Fonts are the least portable aspect of the front panel so always test them on all target platforms.

Colors

With many options for colors, developers commonly encounter the problem of using too many colors. Color can distract the user from important information. For instance, a yellow, green, or bright orange background makes it difficult to see a red danger light. Another problem is that some platforms do not have as many colors available. 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 can have problems detecting the change.

  Also, multiplot graphs and charts can lose meaning when displayed in black and white.

  Use line for plot styles in addition to color.

- Consider coloring the front panel background and objects of user interface VIs with the system colors, or symbolic colors, in the color picker. System colors adapt the appearance of the front panel to the system colors of any computer that runs the VI.
- Use light gray, white, or pastel colors for backgrounds.

  The first row of colors in the color picker contains less harsh colors suitable for front panel backgrounds and normal controls. The second row of colors in the color picker contains brighter colors you can use to highlight important controls. Select bright, highlighting colors only when the item is important, such as an error notification.
- Be consistent in use of color.
- Always check the VI for consistency on other platforms.

Graphics and Custom Controls

You can enhance the functionality of the front panel with imported graphics. You can import bitmaps, Macintosh PICTs, Windows Enhanced Metafiles, and text objects to use as front panel backgrounds, items in pict rings, and parts of custom controls and indicators, as shown in Figure 6-2.

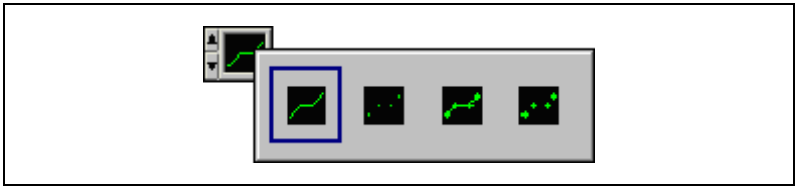


Figure 6-2. Example of Imported Graphics Used in a Pict Ring

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

Refer to the *Porting and Localizing LabVIEW VIs* Application Note for more information about importing graphics on different platforms.

One disadvantage of using imported graphics is that they slow down screen updates. Make sure you do not place indicators and controls on top of a graphic object. That way, the object does not have to be redrawn each time the indicator is updated.

Tip If you must use a large background picture with controls on top of it, divide the picture into several smaller objects and import them separately. Large graphics usually take longer to draw than small ones.

Layout

Consider the arrangement of controls on front panels. Keep front panels simple to avoid confusing the user. For example, you can use menus to help reduce clutter. 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** pull-down menus to create a uniform layout.

Do not overlap controls with other controls, with their label, digital display, or other objects unless you want to achieve a special effect. Overlapped controls are much slower to draw and might flash.

Use decorations, such as a Raised Box or Horizontal Smooth Box, to visually group objects with related functions, as shown in the following figure. Use clusters to group related data. Do not use clusters for aesthetic purposes only.

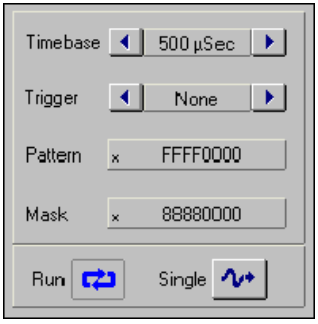


Figure 6-3. Using Decorations to Visually Group Objects Together

For subVI front panels, place the controls and indicators of the subVI so that they correspond to the connector pane pattern. Refer to the *Connector Panes* section of this chapter for more information about connector pane style.

To describe the purpose of the VI or object and to give users instructions for using the VI or object, create descriptions and tip strips for the VI or object. Refer to the *LabVIEW Help* and Chapter 15, *Documenting and Printing VIs*, of the the *LabVIEW User Manual* for more information about creating VI and object descriptions and tips.

Sizing and Positioning

Front panels need to fit on a monitor that is the standard resolution for most intended users. Make the window as small as possible without crowding controls or sacrificing a clear layout. If the VIs are for in-house use and everyone is using high-resolution display settings, you can design large front panels. If you are doing commercial development, keep in mind that some displays have a limited resolution, especially LCD displays and touchscreens.

Front panels should open in the upper-left corner of the screen for the convenience of users with small screens. Place sets of VIs that are often opened together so the user can see at least a small part of each. Place front panels that open automatically in the center of the screen. Centering the front panels makes the VI easier to read for users on monitors of various sizes. Refer to the *LabVIEW Help* for more information about customizing the window appearance.

Labels

The **Context Help** window displays labels as part of the connector pane. If the default value is essential information, place the value in parentheses next to the name in the label. Include the units of the value if applicable. The **Required**, **Recommended**, **Optional** setting for connector pane terminals affects the appearance of the inputs and outputs in the **Context Help** window.

The name of a control or indicator needs to describe its function. If the control is visible to the user, use captions to display a long description and add a short label to prevent using valuable space on the block diagram. For example, when labeling a ring or slide that has options for volts, ohms, or amperes, select an intuitive name for the control. A caption such as “Select units for display” is a better choice than “V/O/A.” Use Property Nodes to change captions programatically.

For Boolean controls, use the name to give an indication of which state corresponds to which function and to indicate the default state. For checkboxes and radio buttons, the user can click the boolean text of the control and the value of the boolean control changes. Free labels next to a Boolean can help clarify the meaning of each position on a switch, as shown in Figure 6-4.

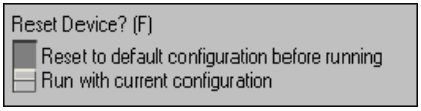


Figure 6-4. Free Labels on a Boolean Control

Paths Versus Strings

When specifying the location of a file or folder, use a path control or indicator. Path controls and indicators work similarly to strings, but LabVIEW formats paths using the standard syntax for the platform you are using. Make sure you set the browse options appropriately on the **Browse** button of file path controls. For example, if the user needs to select a folder, make sure the **Selection Mode** option in the **Browse Options** dialog box is set to **existing dir**.

Use a path constant and the path data type to supply a constant path value to the block diagram. The path constant and data type use the platform-specific notation for paths, unlike the string constant and data type.

Enumerated Type Controls Versus Ring Controls

You cannot change the string labels in an enumerated type control programmatically at run time because the string labels are a part of the data type. When using enumerated type controls, always make a type definition of the control. Creating type definitions prevents you from needing to rewrite the code each time you add or remove an item from an enumerated type control.

Enumerated type controls are useful for making block diagram code easier to read because when you wire an enumerated type control to a Case structure, the string labels appear in the selector label.

Ring controls are useful for front panels the user interacts with, where you want to programmatically change the string labels. You may want to use a ring control instead of a Boolean control because if you decide to change the control to include more than two options, you can add options easily to a ring control.

Default Values and Ranges

Expect the user to supply invalid values to every control. You can check for invalid values on the block diagram or right-click the control and select **Data Range** to set the control item to coerce values into the desired range: **Minimum**, **Maximum**, and **Increment**.

Set controls with reasonable default values. A VI should not fail when run with default values. Remember to show the default in parentheses in the control label for subVI front panels. 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 dialog box. This can save the use of a Boolean switch in many cases.

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, place the interlock logic there. If one or more of the controls are not readily available, you can request the present settings from the instrument to ensure you do not try to set an invalid combination.

Property Nodes

Use Property Nodes to give the user more feedback on the front panel and to make the VI easier to use. The following are examples of using Property Nodes to enhance ease of use:

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

You can modify front panel objects in subVIs using control references. Use the control refnum controls to pass front panel object references to other VIs. After you pass a control reference to a subVI, use Property Nodes and Invoke Nodes to read and write properties and invoke methods of the referenced front panel object. Control references reduce the clutter on the block diagram.

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. Consider including keyboard shortcuts for your VIs even if the use of a mouse is available. Keyboard shortcuts add convenience to a VI.

Pay attention to the key navigation options for buttons on the front panel. Set the tabbing order for the buttons to read left to right and top to bottom. Set the <Enter> key to be the keyboard shortcut for the front panel default control, which is usually the **OK** button. 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 the front panel has a **Cancel** button, set the <Esc> key to be the keyboard shortcut. 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. Select **Edit»Set Tabbing Order** to arrange controls in a logical sequence when the user needs to tab between the controls. For controls that are offscreen, use the **Key Navigation** dialog box to skip over the controls when tabbing or to hide the controls.

Also consider using the key focus property to set the focus programmatically to a specific control when the front panel opens.

Refer to Chapter 4, *Building the Front Panel*, of the *LabVIEW User Manual* for more information about controlling button behavior with key navigation. Refer to the *LabVIEW Help* for more information about setting the tabbing order of front panel objects.

Dialog Boxes

Dialog boxes are an effective way to gather settings and configuration information from the user. Use dialog controls in the dialog box you create to prompt users to configure and set the options for the dialog box. Consider using tab controls, which group front panel controls and indicators in a smaller area, to reduce clutter in a dialog box.

Many modern programs also use dialog boxes to announce messages to the user, but quite often this type of dialog box is overused. Use a status text window to display less serious warnings. Refer to Chapter 4, *Building the Front Panel*, of the *LabVIEW User Manual* for more information about creating dialog boxes.

Block Diagram Style

The block diagram is the primary way for others to understand how a VI works, therefore it is often worth the effort to follow a few simple steps to make block diagrams more organized and easier to read.

Style is as important on the block diagram of the VI as on the front panel. Users may not see the block diagram, but other developers do. A well-planned, consistent block diagram is easier to understand and modify.

Wiring Techniques

You can use the **Align Objects** and **Distribute Objects** pull-down menus on the toolbar to arrange objects symmetrically on the block diagram. When objects are aligned and distributed evenly, you can use straight wires to wire the objects together. Using straight wires makes the block diagram easier to read.

The following are other good wiring tips for keeping the block diagram clean:

- Avoid placing any wires under block diagram objects such as subVIs or structures because LabVIEW can hide some segments of the resulting wire. Draw your wires so that you can clearly see if a wire correctly connects to a terminal.
- Add as few bends in the wires as possible and keep the wires short. Avoid creating wires with long complicated paths because long wires are confusing to follow.
- Delete any extraneous wires.
- Avoid the use of local variables when you can use a wire to transfer data. Every local variable that reads the data makes a copy of the data.
- Do not wire through structures if the data in the wire is not used in the structure.
- Space parallel wires evenly in straight lines and around corners.

Memory and Speed Optimization

There are many things you can do to optimize usage and execution time of a LabVIEW VI. Optimization is important when a program has large arrays or critical timing problems. Refer to the *LabVIEW Performance and Memory Management* Application Note for more information about optimizing your VIs.

Even though optimization can be a very involved topic, you can take the following basic actions to optimize an application:

- If speed is not necessary to a While Loop, add a Wait function to avoid slowing down other tasks. Generally, slowing down other tasks is only an issue with loops that are not very active between iterations, such as user interface loops, because LabVIEW runs the loop as quickly as possible and does not give many processor resources to other tasks. The slowing of tasks outside of the loop results in the computer seeming slow even though it is running a simple loop.

  Adding a slight delay between iterations with the Wait (ms) function can dramatically help the computer run outside tasks normally without affecting the operation of the loop. Typically, a delay of 50 to 100 ms is enough, but other factors in the application can affect the delay. The delay does not require any data dependencies, as shown in Figure 6-5.

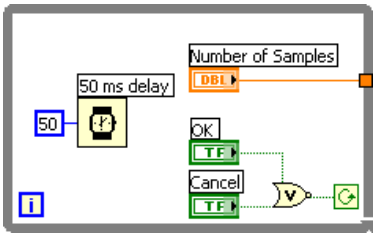


Figure 6-5. While Loop with 50 Millisecond Delay

  Use the Wait Until Next ms Multiple function to time the loop iterations more precisely. Refer to the *LabVIEW Help* for more information about the difference between the Wait function and the Wait Until Next ms Multiple function.

  To avoid this problem completely, use an Event structure. Event structures do not require processor time while waiting for a user interface event.

- If possible, do not build arrays using the Build Array function within a loop because the function makes repetitive calls to the LabVIEW memory manager. A more efficient method of building an array is to use auto-indexing or to pre-size the array and use the Replace Array Element function to place values in it. Similar issues exist when dealing with strings because, in memory, LabVIEW handles strings as arrays of characters.
- Use global and local variables as sparingly as possible. You can use both global and local variables to write VIs very efficiently. However, if you misuse or abuse global and local variables, particularly with data

types, the memory usage of the VI increases and the performance is affected.

Additionally, you can encounter race conditions when reading from and writing to global variables in the same application. These race conditions are difficult to debug since there is no data dependency between different instances of the same global variable on the block diagram.

Consider using functional global variables instead of global variables. Functional global variables do not create extra copies of data and allow certain actions such as initialize, read, write, and empty. Refer to the *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability* Application Note for more information on using functional global variables.

- Choosing the proper array data type for the data to be handled is important in controlling the memory usage of the application. For example, if you have an extended-precision floating-point array of 100,000 values, but the actual values stored in the array are single-precision floating-point values, there is an inefficient use of memory. Using an array of single-precision floating-point values to match the data type stored in the array reduces the memory usage. The following table shows the difference in memory usage.

Array Data Type	Memory Used
Array of 100,000 EXT values	1 MB
Array of 100,000 SGL values	400 KB
Memory Saved	600 KB

- Avoiding coercion dots also can reduce unnecessary memory usage and speed. Coercion dots indicate that LabVIEW converted the value wired to the node to a different representation, which means that LabVIEW made a copy of the data. The effects of coercion dots become more dramatic when you have large arrays of data of inconsistent data types.
- Frequently updating front panel indicators with new data can affect the performance of the VI especially if you are displaying large amounts of data in graphs or charts. To optimize the performance of the VI, display only the necessary information on the front panel and send data to indicators only if the data is different from what the indicator already displays.

Sizing and Positioning

The size of the block diagram window can affect how readable your LabVIEW code is to others. Make the block diagram window no larger than the screen size. In addition to the size of the block diagram window, it is also important to ensure that the LabVIEW code displayed is not too large. Code that is larger than the window is hard to read because it forces users to scroll through the window. If the code is too large to fit on one screen, make sure the user only has to scroll in one direction, horizontal or vertical, to view the rest of the code.

Left-to-Right Layouts

LabVIEW uses a left-to-right layout so block diagrams need to follow this convention. While the positions of program elements do not determine execution order, avoid wiring from right to left. Only wires and structures determine execution order.

Block Diagram Comments

Developers who maintain and modify VIs need good documentation on the block diagram. Without it, modifying the code is more time-consuming and error-prone. The following are some suggestions for commenting the block diagram. Refer to the *LabVIEW Help* for more information about creating free labels.

- Use comments on the block diagrams to explain what the code is doing. LabVIEW code is not self-documenting even though it is graphical. The free label located on the **Functions** palette has a colored background which works well for block diagram comments.
- Do not show labels on function and subVI calls because they tend to be large and unwieldy. A developer looking at the block diagram can find the name of a function or subVI by using the **Context Help** window.
- Use free labels on wires to identify their use. Labeling wires is useful for wires coming from shift registers and for long wires that span the entire block diagram.
- Use labels on structures to specify the main functionality of that structure.
- Use labels on constants to specify the nature of the constant.
- Use labels on Call Library Function Nodes to specify what function the node is calling and the path to the library being called.

- Use comments to document algorithms that you use on the block diagrams. If you use an algorithm from a book or other reference, provide the reference information.

Call Library Function Nodes and Code Interface Nodes

VIs that contain Call Library Function Nodes and Code Interface Nodes (CINs) are platform specific. VIs that call shared libraries using a Call Library Function Node are platform dependent and only function on different platforms if you have a corresponding library for each platform. If you move a VI that contains a Call Library Function Node to another platform, you must update the node to specify the shared library compiled for the new platform.

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. When you move VIs that contain CINs to other platforms, you must recompile the code for the new platform and reload it into the CIN. For this reason, only use CINs if they provide functionality not available with shared libraries.

If a VI contains a Call Library Function Node or a CIN and you want to use that VI on multiple platforms, consider creating different versions of the VI that contain the Call Library Function Node and the CIN for each platform.

Type Definitions

Use a type definition when you use the same unique control in more than one location or when you have a very large data structure passing between several subVIs. By using a type definition control, LabVIEW automatically propagates changes to the control or the data structure throughout all the VIs and subVIs.

Refer to the *LabVIEW Custom Controls, Indicators, and Type Definitions* Application Note for more information about using type definitions.

Sequence Structures

Use sequence structures sparingly since they hide code. Try to rely on data flow rather than sequence structures to control the order of execution. With sequence structures, you break the left to right data flow paradigm whenever you use a sequence local. To help control data flow, you can use error clusters.

If you must use a sequence structure, consider using a Flat Sequence structure so you do not hide any code. If you use a Stacked Sequence structure in the VI, save the VI with the most important frame showing.

Refer to Chapter 8, *Loops and Structures*, of the *LabVIEW User Manual* for more information about using sequence structures.

Icon and Connector Pane Style

Using good style techniques when creating the icons and connector panes for VIs can greatly benefit users of those VIs.

Icons

The icon represents the VI on a palette and a block diagram. When subVIs have well-designed icons, developers can have a better understanding of the subVI without the need for excessive documentation.

Use the following suggestions when creating icons:

- Create a meaningful icon for every VI. The LabVIEW libraries are full of well-designed icons that try to show the functionality of the underlying program; use them as prototypes where applicable. When you do not have a picture, text is acceptable. If you localize the application, make sure you also localize the text on the icon.

Tip A good size and font choice for text icons is 8 point Small Fonts in all caps.

- Do not use colloquialisms when making an icon. Colloquialisms are difficult to translate. Users whose native language is not English may not understand what the icon does from a picture that does not translate well. For example, do not represent a datalogging VI with a picture of a tree branch or a lumberjack.
- Create a unified icon style for related VIs. This helps users visually understand what subVI calls are associated with the top-level VI.
- Always create a black and white icon for printing purposes. Not every user has access to a color printer.
- Always create standard size (32 × 32 pixels) icons. VIs with smaller icons can be awkward to select and wire. Smaller icons also tend to look strange when wired.

- Make sure the subVI icon is visible on the block diagram. Right-click the subVI and select **Visible Items»Terminals**. Make sure there is no checkmark next to the **Terminals** options. The subVI icon is more helpful than the connector pane on the block diagram.

Example of Intuitive Icons

The Report Generation VIs are examples of icons designed with good style techniques.

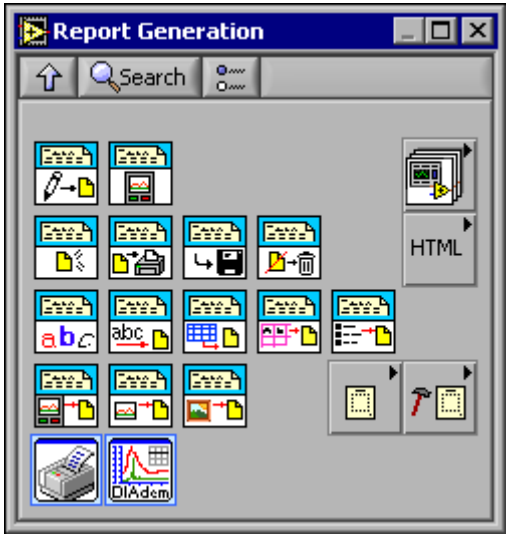


Figure 6-6. Report Generation VIs

The Report Generation VIs use images of a disk, a printer, a pencil, and a trashcan to represent what the VIs do. The image of a piece of paper with text on it represents a report and is a common element in the icons. This consistency unifies the icon designs. Notice that none of the icons are language dependent, thus they are suitable for speakers of any language.

Refer to Chapter 7, *Creating VIs and SubVIs*, of the *LabVIEW User Manual* for more information about creating icons.

Connector Panes

A connector pane is the set of terminals that correspond to the controls and indicators of a VI. Refer to Chapter 7, *Creating VIs and SubVIs*, of the *LabVIEW User Manual* for more information about setting up connector panes.

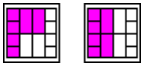
Use the following suggestions when creating connector panes:

- Always select a connector pane pattern with extra terminals. Unforeseen additions to the VI may require more terminals. Changing patterns requires relinking to the subVI in all calling VIs and may result in wiring incompatibilities.

  A good connector pane pattern is the $4 \times 2 \times 2 \times 4$ pattern, shown at left.

- Wire inputs on the left and outputs on the right to follow the standard left-to-right data flow.

  When assigning terminals as inputs and outputs, make sure to split the terminals of the connector pane consistently. If you need to use the middle four terminals of the $4 \times 2 \times 2 \times 4$, divide them either horizontally or vertically. For example, either assign the inputs to the top two terminals and the outputs to the bottom two terminals, or assign the inputs to the left two terminals and the outputs to the right two terminals, shown at left.

- Choose the same connector pane pattern for related VIs. Wire related inputs and outputs on opposite terminals so that references, taskIDs, error clusters, and other common terminals line up correctly.
- Try to choose only connector pane patterns with 16 or fewer terminals. While patterns with more terminals might seem useful, they are very difficult to wire. If you need to pass more data, use clusters.
- The connector pane should have a minimum of eight terminals, using the 4×4 pattern, shown at left. This pattern ensures that all VIs, even VIs with few inputs, line up correctly and have straight wires connecting them.

- When assigning terminals, keep in mind how the VIs will be wired together. If you create a group of subVIs that you use together often, give the subVIs a consistent connector pane with common inputs in the same location to help you remember where to locate each input. If you create a subVI that produces an output another subVI uses as the input, align the input and output connections to simplify the wiring patterns.
- Use the **Required**, **Recommended**, **Optional** setting for every terminal to prevent users from forgetting to wire subVI connections.

Use required for inputs that must be wired for the subVI to run properly. Use optional for inputs that have default values that are appropriate for the subVI most of the time.

- Include **error in** and **error out** clusters on all subVIs, even if the subVI does not process errors. **Error in** and **error out** clusters are helpful for controlling execution flow. If a subVI has an incoming error, you can use a Case structure to send the error through the VI without executing any of the subVI code.
- Strive for similar arrangements between panel and connector pane layouts.

Refer to the *LabVIEW Help* for more information about creating connector panes.

Style Checklist

Use the following checklists to help you maintain consistent style and quality. You can copy these checklists to use on all LabVIEW projects. You also can add, remove, or modify any of the following guidelines to customize these checklists to fit the specifications of your project.

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.
- ❑ Use **Tools»Edit VI Library** to mark top-level VIs within an LLB.
- ❑ Create a `.mnu` file, if the VIs will be used as subVIs. Be sure to arrange the palettes, name the menus, and hide dependent subVIs.
- ❑ Give VIs meaningful names without special characters, such as backslash (`\`), slash (`/`), colon (`:`), and tilde (`~`).
- ❑ Use standard extensions so the operating system can distinguish files (`.vi`, `.ctl`).
- ❑ Capitalize first letters of VI names.

- ❑ Distinguish example VIs, top-level VIs, subVIs, controls, and global variables by saving them in subdirectories or separate libraries in the same directory, or by giving them descriptive names such as `X Main.vi`, `X Example.vi`, `X Global.vi`, and `X TypeDef.ctl`.
- ❑ Write a VI description, proofread it, and check the **Context Help** window.
- ❑ Include your name and/or company and the date in the **VI Description** on the **Documentation** page of the **VI Properties** dialog box.
- ❑ Create a meaningful black and white icon in addition to a color icon.
- ❑ Choose the $4 \times 2 \times 2 \times 4$ connector pane pattern to leave extra terminals for later development. Use a consistent layout across related VIs.
- ❑ Avoid using connector panes with more than 16 terminals.
- ❑ Use **Required**, **Recommended**, and **Optional** settings on the connector pane.
- ❑ Set print options to display the 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.
- ❑ Avoid using VIs that are platform specific if the VIs need to run on multiple platforms.
- ❑ Consider creating different versions of VIs that contain platform specific VIs and functions.

Front Panel Checklist

- ❑ Give controls meaningful names. Use consistent capitalization.
- ❑ Make the background of control and indicator labels transparent.
- ❑ Check for consistent placement of control names.
- ❑ Use standard, consistent fonts—application, system, and dialog—throughout all front panels.

- ❑ Use **Size to Text** for all text for portability, and add carriage returns if necessary.
- ❑ Use path controls instead of string controls to specify the location of files or directories.
- ❑ Put default values in parentheses on captions for controls in subVI front panels.
- ❑ Include unit information in names. If applicable, for example, use **Time Limit (10 Seconds)**.
- ❑ Write descriptions for controls and indicators, including array, cluster, and refnum elements. Remember that you might need to change the description if you copy the control.
- ❑ Create tip strips for front panel items so users have an easy way to understand the functionality of controls on user interface VIs.
- ❑ Arrange controls logically. For user interface 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 pane terminals.
- ❑ Arrange controls attractively, using the **Align Objects** and the **Distribute Objects** pull-down menus.
- ❑ Do not overlap controls.
- ❑ Use color logically and sparingly, if at all.
- ❑ Provide a **Stop** button if necessary. Do not use the **Abort** button to stop a VI. Hide the **Abort** button.
- ❑ Use ring controls and enumerated controls where appropriate. If you are using a Boolean control for two options, consider using an enumerated control instead to allow for future expansion of options.
- ❑ Use custom controls or typedefs for common controls, especially for rings and enums.
- ❑ Label custom controls with the same name as the file. For example, `Alarm Boolean.ctl` has the default name Alarm Boolean.

- ❑ Make sure that all the controls on the front panel are of the same style. For example, do not use both classic and 3D controls on the front panel.

Block Diagram Checklist

- ❑ Avoid creating extremely large block diagrams. Limit the scrolling necessary to see the entire block diagram to one direction.
- ❑ Label controls, important functions, constants, Property Nodes, local variables, global variables, and structures.
- ❑ Add comments. Use object labels instead of free labels where applicable and scrollable string constants for long comments.
- ❑ 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.
- ❑ Flow data from left to right. Wires enter from the left and exit to the right.
- ❑ Align and distribute functions, terminals, and constants.
- ❑ Label long wires with small free labels with white backgrounds.
- ❑ Avoid placing block diagram objects, such as subVIs or structures, on top of wires.
- ❑ Do not wire behind objects.
- ❑ Use path constants instead of string constants to specify the location of files or directories.
- ❑ Make good use of reusable, testable subVIs.
- ❑ Use error in and error out clusters in all 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.

- ❑ Consider using a Flat Sequence structure if you have a sequence structure in the VI.
- ❑ Save the VI with the most important frame of multiframed structures (Case, Stacked Sequence, and Event structures) showing.
- ❑ Review the VI for efficiency, data copying, and accuracy, especially parts without data dependency.
- ❑ Make sure the subVI icon, rather than the connector pane, is visible on the block diagram.
- ❑ Use a type definition when you use the same unique control in more than one location or when you have a very large data structure passing between several subVIs.
- ❑ If you open references to a LabVIEW object, such as an application, control, or VI, close the references by using the Close LV Object Reference function. Otherwise, the reference stays in memory even though the VI no longer uses it.
- ❑ Make sure the **Name Format** for Property Nodes and Invoke Nodes is set to **Short Names** to ensure the best readability of the block diagram.
- ❑ Place any control on the connector pane of a subVI on the left side of the block diagram.
- ❑ When designing a subVI, make sure control and indicator terminals on the connector pane are not insides structures on the block diagram.

A

References

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

Dynamics of Software Development. Jim McCarthy, Microsoft Press. 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.

Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products. Daniel P. Freedman and Gerald M. Weinberg, Dorset House Publishing Co., Inc. A 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. Describes what is expected by ISO 9001 in conjunction with ISO 9000.3 and provides templates for documentation.

LabVIEW Technical Resource. Edited by Lynda P. Gruggett, LTR Publishing, phone (214) 706-0587, fax (214) 706-0506. `http://www.LTRPub.com`. A quarterly newsletter and disk of VIs that features technical articles about all aspects of LabVIEW.

Microsoft Secrets. Michael A. Cusumano and Richard W. Selby, Free Press. 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.

Rapid Development: Taming Wild Software Schedules. Steve C. McConnell, Microsoft Press. Explanation of software engineering practices with many examples and practical suggestions.

Software Engineering. Edited by Merlin Dorfman and Richard Thayer, IEEE Computer Science Press. Collection of articles on a variety of software engineering topics, including a discussion of the spiral life cycle model by Barry W. Boehm.

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

Software Engineering Economics. Barry W. Boehm, Prentice-Hall. Description of the Wideband Delphi and COCOMO estimation techniques.

B

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at ni.com/support. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, hardware schematics and conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
  - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting ni.com/ask. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit ni.com/custed for self-paced tutorials, videos, and interactive CDs. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Glossary

B

black box testing	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)	Model for judging the maturity of the processes of an organization and for identifying the key practices required to increase the maturity of these processes. The Software CMM (SW-CMM) is 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.
COCOMO Estimation	COnstructive COst MOdel. A formula-based estimation method for converting software size estimates to estimated development time.
code and fix model	Lifecycle model that involves developing code with little or no planning and fixing problems as they arise.
configuration management	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 you can rebuild a particular release of software, if necessary. Configuration management implies archival development of tools, source code, and so on.

F

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

functional global variable	A VI containing a single-iteration While Loop with an uninitialized shift register. A functional global variable is similar to a global variable, but it does not make extra copies of data in memory. Functional global variables also can expand beyond simple read and write functionality.

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	Model for software development, including steps to follow from the initial concept through the release, maintenance, and upgrading of the software.

S

Software Engineering Institute (SEI)	Federally funded research and development center 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 the Software Engineering Institute Web site at `www.sei.cmu.edu` for more information about the institute.
source lines of code	Measure of the number of lines of code that make up a text-based 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	Lifecycle model that emphasizes risk management through a series of iterations in which risks are identified, evaluated, and resolved.
stub VI	Nonfunctional prototype of a subVI. A stub VI has inputs and outputs, but is incomplete. Use stub VIs during early planning stages of an application design as a place holder for future VI development.
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	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, use white box testing to verify all the paths of execution of the module have been exercised.
Wideband Delphi estimation	Technique for a group to use 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