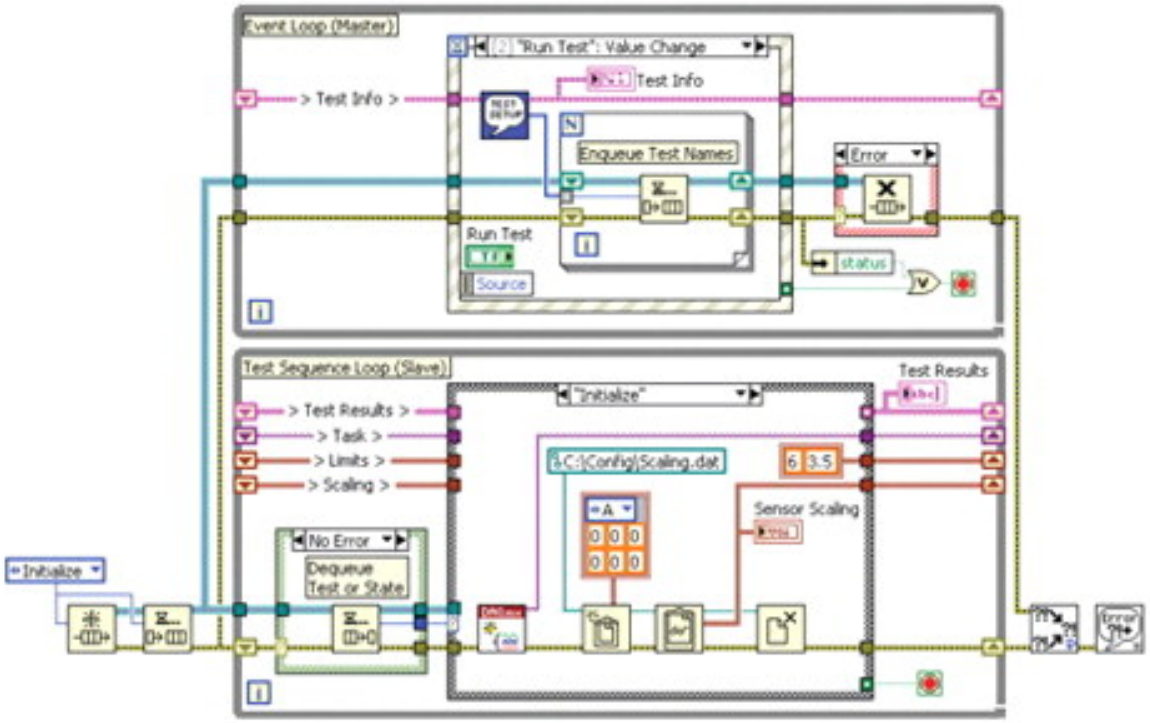
LabVIEW

PRENTICE HALL



The LabVIEW Style Book

Peter A. Blume

EASE OF USE • EFFICIENCY • READABILITY • SIMPLICITY
PERFORMANCE • MAINTAINABILITY • ROBUSTNESS

Table of Contents

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

Prepare for Good Style

2

LabVIEW style is positively influenced by preparing to develop your applications before you begin coding. First and foremost, write a requirements specification. It is much easier to develop good code when working from a specification instead of deriving the requirements on-the-fly. Second, design the software. Combine standard software design principles, such as the State Machine Diagram, with LabVIEW best practices, such as the State Machine design pattern. Third, configure the LabVIEW environment. Make use of desired preferences and previously developed code, such as reusable templates, instrument drivers, and utilities. Finally, establish some conventions for organizing, naming, and controlling your project files. An organized project spawns organized source code.

This chapter describes preparation techniques that are conducive to good development style. It is noteworthy that several of these techniques relate equally to configuration management (CM) as well as style. Indeed, many of the Rules in this chapter, as well as throughout the book, may complement or extend a CM process. All topics are presented in the context of style, which is one element of CM. Due to the wide diversity of LabVIEW applications and target industries, there is no universal CM system that can work for most organizations. Therefore, the reader may evaluate the relevance of these materials to their own CM system.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

2.1 Specifications

Most LabVIEW applications begin with a business objective that involves an industrial measurement and automation challenge. Business objectives include accelerating research and development, increasing production throughput, improving quality, and reducing manual labor. Most people use LabVIEW because it is the fastest method of developing applications for achieving their desired business objective. There are many other benefits people consider, such as the graphical paradigm, open support for thousands of instruments, platform portability, and network connectivity. However, most industrial decisions are justified by considering the savings of time and money. As such, most industrial users choose LabVIEW because of the rapid software development cycle.

LabVIEW provides the opportunity to develop applications very quickly. High-level tools such as Express VIs, instrument drivers, templates, and examples empower the user to develop software with very little planning or preparation. The speed with which you can connect to one or more instruments and begin making measurements is extraordinary. Additionally, rapid code development provides developers and managers, often under significant time pressures, instant gratification. As a result, many LabVIEW developers adopt fast programming habits. Shortcuts are routinely taken. Any development steps that are deemed unnecessary are frequently skipped.

Requirements specifications development constitutes a critical phase of the application development cycle. Absence of a specification leads to significant time and effort expended developing software that may not satisfy the intended objective. Alternatively, the requirements that are understood today may not be the same as what is required tomorrow—a phenomenon known as *scope creep*. Moreover, how do we know when our application is complete without documented specifications?

In practice, I would estimate that fewer than 25% of LabVIEW developers write formal requirements specifications. Most projects are initiated in meetings in which hardware and overall system requirements dominate the agenda. Hardware seems to take priority over software because the products, features, part numbers, prices, and lead times seem more tangible and inflexible. Since LabVIEW provides unlimited flexibility with respect to software functionality, the features, prices, and lead times of a LabVIEW application appear virtual or even self-imposed. In fact, every software feature has a lead time and price in terms of development hours. This simple fact is often overlooked or underestimated. As a result, misunderstood requirements and scope creep abound throughout the LabVIEW community.

Scope creep is not purely evil and forbidden. In R&D environments in particular, measurement and automation requirements often change naturally as the research emphasis, or product or process design evolves. LabVIEW's flexibility and rapid development capability are advantageous in such dynamic industrial settings. Alternatively, if the original requirements are satisfied ahead of schedule, then the project scope may expand to include greater levels of automation. This correlates to greater benefits. However, if the specifications are not documented, then it becomes very difficult to verify what the requirements are at any given point in time throughout the project life cycle, whether you are succeeding, whether the application scope has expanded, and most importantly, when the project is complete.

Theorem 2.1: *Written specifications positively influence LabVIEW style.*

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

The presence or absence of a written specification has a direct effect on LabVIEW programming style, as well as the overall outcome of the application. A specification is a detailed description of the project's requirements. Writing a specification entails significantly more planning and attention to detail than verbal communications. It is a much different thought process than source code development. It forces the writers and readers to envision the entire scope of the application before the software development begins. The specification may be reviewed and refined by the contributors as much as necessary to build consensus. Once the coding begins, the risk for misunderstandings, oversights, and unforeseen scope change is dramatically reduced.

Reviewing or contributing to the specification is an excellent exercise for a developer preparing to design the application. During all meetings I attend, I always strive to understand the project's requirements as thoroughly as possible. No matter how confident I may feel at meeting's end, I have never proceeded to sit down and write a specification without new questions and ideas arising in the process. Following up on these ideas translates to better understanding and agreement of project scope, which translates to a better LabVIEW application, and more satisfied end users.

Theorem 2.2: *Unforeseen scope changes hinder good style.*

Unanticipated expansion of scope can hinder good style, and in the most severe instances may even diminish an application into spaghetti. A developer may choose to begin with the simplest architecture that satisfies the application's initial requirements, in order to expedite development. For example, a simple top-level architecture consisting of a single While Loop may be chosen. This architecture is often not amenable to an application's expanded requirements. When the scope changes substantially, the While Loop is expanded to accommodate more nodes and wires and may become large and unwieldy. Also, as the diagram's node quantity increases, there are more obstacles preventing neat placement of additional nodes, wires, and structures, resulting in a sloppy diagram. Indeed, this may have happened in the Spaghetti VI example from Chapter 1, "The Significance of Style." If you take a second look at Figure 1-3, you will see a single While Loop containing many functions, subVIs, wires, terminals, and Case structures. This is an application that *might* have started out neat at one time and evolved, as the work scope evolved, into spaghetti. The architecture might have been appropriate for the original requirements but was inappropriate for the eventual requirements. Realistically, if this were the case, the developer certainly should have changed the architecture, among many other possible improvements, after the work scope changed. It is also likely that the application was developed from the outset without a fundamental understanding of the requirements and without a style convention. In any event, the process of writing and reviewing a specification significantly reduces the potential for large unexpected changes of scope, reducing the extent of corresponding source code revisions, thereby improving style.

If a specification document does not exist, either the developer does not fully understand the requirements or they reside in the developer's memory. The two situations are very similar. In the former situation, the software will require substantial modifications as the developer's understanding evolves. This is also known as the Code and Fix software life cycle development model. The more overall edits there are, including changes because of misunderstood requirements, the messier the source code can become, per Theorem 2.2. This is generally true with all programming languages.

In the latter situation, in which the requirements are understood but not documented, several problems can arise. Skipping the documentation reduces the amount of planning and consideration, which can result in an inferior initial release that requires many revisions. Hence, the Code and Fix life cycle model still applies. Additionally, the memory-resident specification might become the

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

developer's exclusive intellectual property. Inevitably, the developer will forget some of the details, and her ability to maintain the application decays over time. More concerning is that the developer might not be available to support the application throughout its life cycle. Most people change responsibilities, jobs, companies, and careers, and LabVIEW developers are no exception.

Speaking from experience, Bloomy Controls routinely receives inquiries from companies that need to maintain, fix, or refactor their legacy LabVIEW applications. When the specifications and documentation are lacking, the source code is often sloppy. These applications are generally rewritten from scratch, beginning with a specification. Conversely, when good specifications exist, the source code is predictably high quality—not always, but most of the time.

2.1.1 Best Practices for Specifications Development

Good specifications originate as manually recorded notes. Many of the meetings in which projects are specified resemble fast-paced brainstorming sessions, and high-speed diligent note taking is essential. Bound notepads are a standard method for professional engineers and scientists to track their daily activities. They are also an ideal method of initially capturing raw specifications.

Rule 2.1 Maintain a LabVIEW project journal

The bound notepad serves as a LabVIEW project journal. Record the date of each activity, and maintain your project-related notes in chronological order. As notes are appended throughout the project's life cycle, they need not be limited to requirements; notes are commonly extended to include design notes and maintenance logs. The notes can also include bug descriptions, wish lists, data, manual calculations, notes from support inquiries, and artistic doodles. The function of a project journal for a LabVIEW developer is similar to a laboratory journal for a scientist or inventor. Laboratory journals are used to preserve the experimental data and observations that are part of any scientific investigation. They are the basis for further analysis, discussion, evaluation, and interpretation[1]. Similarly, project journals contain any mixture of specifications and data that form the basis of the software design. In many situations, scientists and inventors use LabVIEW to conduct their experiments. Indeed, many scientists and inventors are also LabVIEW developers. Conversely, LabVIEW developers support the scientific method when we record our activities in a project journal. In some cases, the data that is recorded might even supplement the proof of discovery or invention when a company applies for a patent.

Recording the raw specifications in a project journal constitutes a good specification-development practice. Per Theorem 2.1, specifications positively influence LabVIEW style. The developer can refer to her notes as needed throughout the development and maintenance of the application, as long as she remains available to do so and does not lose or destroy the notepad. Moreover, project journals form the basis of a more formal requirements specification.

Simplicity and speed are the primary advantages of project journals, yet there are several limitations:

1. Bound notepads provide minimal organization. Project notes are entered chronologically. Inactive requirements, ideas, and notes are often mixed with active ones. It is often difficult to distinguish active requirements from inactive ones.
2. Many people do not have clear, legible handwriting. Indeed, it is very difficult for even the neatest writers to maintain legibility throughout a rapid brainstorming session, the kind of environment in which many LabVIEW projects are specified.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

3. Notepads are not transferable. It is inefficient, at best, for multiple project team members to share someone's handwritten notes as project reference materials. Instead, notepads function more like a personal hard copy of one individual's daily activities related to a project.
4. If the raw notes are not formalized and expounded upon, the composer does not gain new insights and perspectives. Moreover, it is not possible to receive feedback, build consensus, and confirm the requirements among multiple project team members if the project team does not share and review the requirements.

Rule 2.2 Write a requirements specification document

Compose a requirements specification document for each application, using the raw notes from the project journal as a reference. A requirements specification is a statement of the project's primary objectives and a prioritized listing of the required features. It contains sufficient detail to clearly describe all important requirements that contribute to the objectives. It should contain very few design and implementation details. Hence, the requirements specification should not place constraints on *how* the project is developed, except where these constraints affect the project's outcome.

The Institute of Electrical and Electronics Engineers (IEEE) offers a Recommended Practice for Software Requirements Specifications, standard 830-1998, and a Guide for Developing System Requirements Specifications, standard 1233-1998. These are excellent references for writing software and system specifications that apply to any programming language, industry, and application type. Specifications written according to these standards are very thorough, high-quality documents.

In some evolutionary industries, the rapid pace of changes or time-to-market pressures pose challenges to the process of writing formal requirements specification documents. In many situations, it is desirable to have the software developed in parallel with product development. Many of the performance specifications that affect the application's requirements are unknown or change frequently during application development. These situations do not justify abandoning the requirements specification process. Instead, a very aggressive specification-development process should be pursued in parallel with product and software development. Specifically, Agile project-management methods such as Extreme Programming are best suited to evolutionary projects. At Bloomy Controls, we dedicate a resource to the specifications-development and -maintenance effort, separate from the software, to gather and maintain the requirements in an iterative manner. The requirements specification need not finalize requirements that are evolving. Instead, try to determine most-probable and worst-case scenarios for the undefined requirements, and include them in the specification. If this is not feasible, simply use "TBD" and come back to it later. In many government-regulated industries, however, TBDs are expressly forbidden. New projects are formed exclusively to resolve the unknown requirements before the software development commences. However, government-regulated industries are generally not rapidly evolving.

In summary, the LabVIEW project journal contains an informal history of the project requirements and evolution, and the requirements specification document establishes the active project scope. These elements are prerequisites for good programming style.

2.1.2 LabVIEW Project Requirements Specification

LabVIEW projects represent a specialized subset of the vast software industry for which the IEEE standards were created. Specifically, LabVIEW is primarily applied to measurement and automation applications, and a relatively fast development cycle is expected. Most applications perform

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

acquisition, analysis, and presentation. These factors can be used to form a specialized standard for writing requirements specifications for LabVIEW projects. Specifically, I recommend the following elements:

- Statement of high-level objectives
- Budget
- Timetable
- Detailed requirements for acquisition, analysis, and presentation
- Priorities for each requirement
- Test methodology

Figure 2-1 shows a LabVIEW Project Requirements Specification template.

The high-level objectives should clearly describe why the project is being undertaken. It is particularly useful to describe how the objectives relate to the company's core products and services. This is also known as the business objective. For example, a high-level objective might be stated as follows:

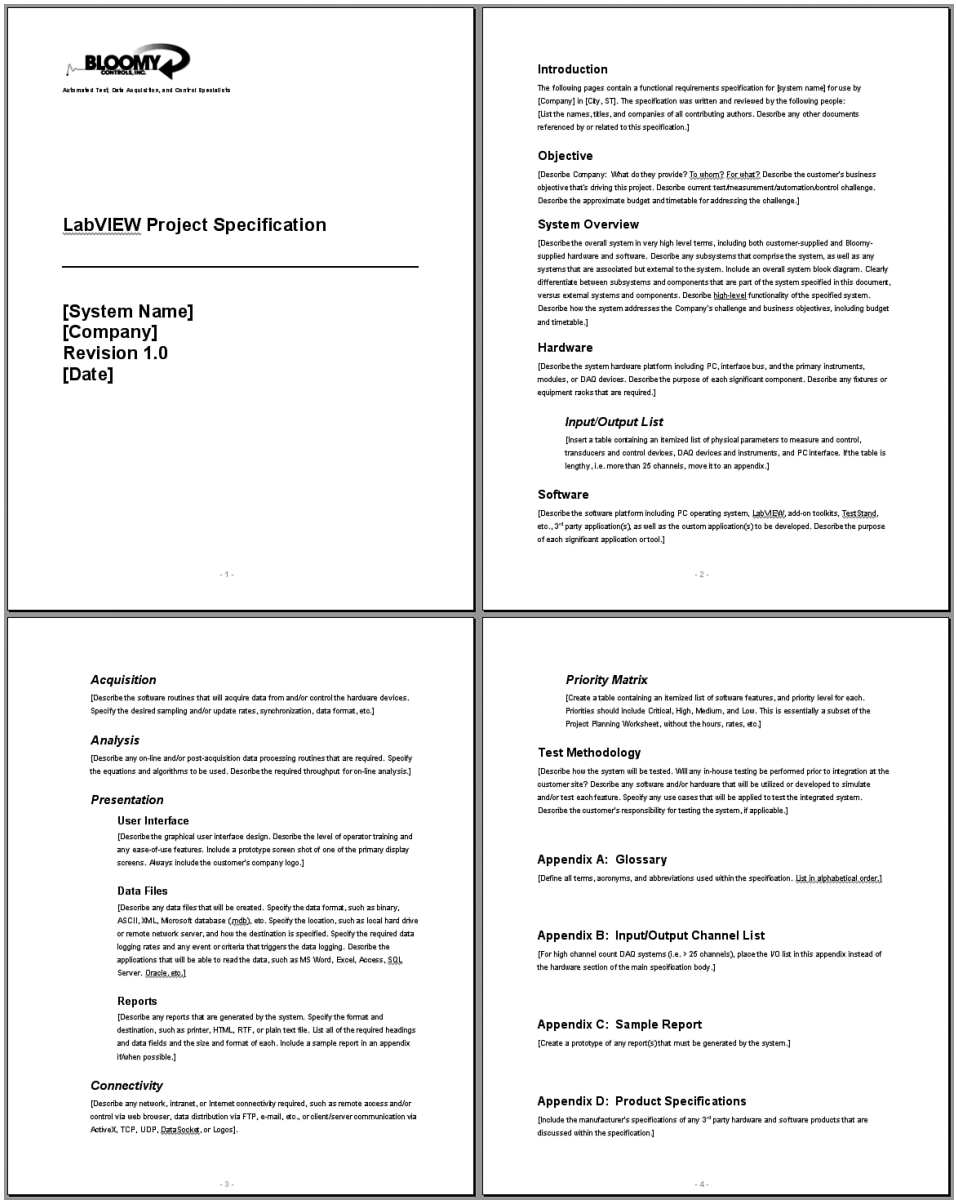
> Evaluate a low-emissions exhaust system that is being considered by the automotive industry for 2010 model automobiles. The system must comprehensively test the first 100 prototypes in January 2008. The system must be scaled into high-volume production in July. The market potential of the product is estimated to be $1 billion over a 5-year span.

Budget and timetable are very important considerations that not only directly affect the system design, but might also determine the overall feasibility. It is extremely important to get an idea of how much funding is available and the required completion date at the outset of the project. It is common to identify significant obstacles at this point. For example, you might find that the lead time or expense of some of the required elements exceeds the requirements. The earlier in the project that you identify such discrepancies, the more time and effort you will save.

I like to organize the requirements into four primary categories: acquisition, analysis, presentation, and test methodology. The **acquisition** section describes the required measurement and control hardware interfaces. I find it useful to create a spreadsheet that itemizes every physical parameter or input and output (I/O) point that must be measured or controlled, including the engineering units, range, accuracy, and rate at which the parameters must be acquired or generated. This information can then be used to specify transducers, signal conditioning, digitizers, instruments, and control devices. These choices are then entered right into the same spreadsheet, providing further definition of the system. Note that the hardware interfaces and I/O requirements help determine the scope of the LabVIEW development required to access these interfaces, such as readily available instrument drivers, lower-level development using VISA or DAQmx, or simple interactive configuration using Express VIs.

The **analysis** section describes any mathematical processing that is required, either online as data is acquired or offline using data files and a separate routine. Some types of analysis include data reduction or decimation, corrections, digital filters, peak searches, limit evaluations, statistics, and control algorithms. For example, statistical process control (SPC) is very common in automotive manufacturing applications. SPC entails computing such parameters as mean, range, control limits, standard deviation, process capability, and more, and graphically displaying the results as Xbar & R and other charts. The analysis section of the requirements specification describes such calculations.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27



BLOOMY CONTROLS INC.
Automated Test, Data Acquisition, and Control Specialists

LabVIEW Project Specification

[System Name]
[Company]
Revision 1.0
[Date]

- 1 -

Introduction

The following pages contain a functional requirements specification for [system name] for use by [Company] in [City, ST]. The specification was written and reviewed by the following people:
[List the names, titles, and companies of all contributing authors. Describe any other documents referenced by or related to this specification.]

Objective

[Describe Company: What do they provide? To whom? For what? Describe the customer's business objective that's driving this project. Describe current test/measurement/automation/control challenge. Describe the approximate budget and timetable for addressing the challenge.]

System Overview

[Describe the overall system in very high level terms, including both customer-supplied and Bloomy-supplied hardware and software. Describe any subsystems that comprise the system, as well as any systems that are associated but external to the system. Include an overall system block diagram. Clearly differentiate between subsystems and components that are part of the system specified in this document, versus external systems and components. Describe high-level functionality of the specified system. Describe how the system addresses the Company's challenge and business objectives, including budget and timetable.]

Hardware

[Describe the system hardware platform including PC, interface bus, and the primary instruments, modules, or DAQ devices. Describe the purpose of each significant component. Describe any fixtures or equipment racks that are required.]

Input/Output List

[Insert a table containing an itemized list of physical parameters to measure and control, transducers and control devices, DAQ devices and instruments, and PC interface. If the table is lengthy, i.e. more than 25 channels, move it to an appendix.]

Software

[Describe the software platform including PC operating system, LabVIEW, add-on toolkits, TestStand, etc., 3rd party application(s), as well as the custom application(s) to be developed. Describe the purpose of each significant application or tool.]

- 2 -

Acquisition

[Describe the software routines that will acquire data from and/or control the hardware devices. Specify the desired sampling and/or update rates, synchronization, data format, etc.]

Analysis

[Describe any on-line and/or post-acquisition data processing routines that are required. Specify the equations and algorithms to be used. Describe the required throughput for on-line analysis.]

Presentation

User Interface

[Describe the graphical user interface design. Describe the level of operator training and any ease-of-use features. Include a prototype screen shot of one of the primary display screens. Always include the customer's company logo.]

Data Files

[Describe any data files that will be created. Specify the data format, such as binary, ASCII, XML, Microsoft database (.mdb), etc. Specify the location, such as local hard drive or remote network server, and how the destination is specified. Specify the required data logging rates and any event or criteria that triggers the data logging. Describe the applications that will be able to read the data, such as MS Word, Excel, Access, SQL Server, Oracle, etc.]

Reports

[Describe any reports that are generated by the system. Specify the format and destination, such as printer, HTML, RTF, or plain text file. List all of the required headings and data fields and the size and format of each. Include a sample report in an appendix if/when possible.]

Connectivity

[Describe any network, intranet, or Internet connectivity required, such as remote access and/or control via web browser, data distribution via FTP, e-mail, etc., or client/server communication via ActiveX, TCP, UDP, DataSocket, or Logos].

- 3 -

Priority Matrix

[Create a table containing an itemized list of software features, and priority level for each. Priorities should include Critical, High, Medium, and Low. This is essentially a subset of the Project Planning Worksheet, without the hours, rates, etc.]

Test Methodology

[Describe how the system will be tested. Will any in-house testing be performed prior to integration at the customer site? Describe any software and/or hardware that will be utilized or developed to simulate and/or test each feature. Specify any use cases that will be applied to test the integrated system. Describe the customer's responsibility for testing the system, if applicable.]

Appendix A: Glossary

[Define all terms, acronyms, and abbreviations used within the specification. List in alphabetical order.]

Appendix B: Input/Output Channel List

[For high channel count DAQ systems (i.e. > 25 channels), place the I/O list in this appendix instead of the hardware section of the main specification body.]

Appendix C: Sample Report

[Create a prototype of any report(s) that must be generated by the system.]

Appendix D: Product Specifications

[Include the manufacturer's specifications of any 3rd party hardware and software products that are discussed within the specification.]

- 4 -

Figure 2-1
A template for a LabVIEW Project Requirements Specification[2]

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

Presentation includes all human-readable interfaces to the data that the application generates. This includes the graphical user interface, ASCII data files, and reports. One technique that is extremely useful and very easy in LabVIEW is creating a prototype user interface and including one or more screenshots in your specification. Do not spend a lot of time building an actual working prototype or experimenting with fonts and colors at this point; just apply enough controls and indicators on a front panel to illustrate how the user interface might appear. This is a powerful technique for revealing requirements that might have been overlooked or misunderstood. As the saying goes, a picture is worth a thousand words. Likewise, it is beneficial to prototype a sample report or data file using Microsoft Word or Excel.

Note that prototyping candidate display screens and reports generally entails some design effort, which falls outside the pure definition of a requirements specification. In an ideal world, a solid boundary exists between the specification and design phases. However, in practice, prototype designs are instrumental in fleshing out some of the functional requirements. In addition, the requirements specification should be viewed as a living document that is revised as necessary throughout the design and development phases. Therefore, the level of detail contained within the requirements specification might evolve with time. Indeed, a requirements specification might eventually form the basis of a user manual. Be sure to save each revision, or you might lose sight of the original requirements and fall back into the Code and Fix development model. Source code control tools ensure that each revision of the project documents, as well as source code, is automatically archived in a repository.

The **test methodology** describes how the system is tested. A test scenario is conceived that verifies the correct operation of the software and hardware. Ideally, every required feature has a corresponding test that isolates the feature and verifies that it functions properly. Additionally, a series of use cases can be performed in which the most common uses of the software are defined and executed under controlled conditions. For a test and measurement application, it is common to apply a unit under test (UUT) with known characteristics and compare the application's measurements with the known values. However, in many real-world projects, a characterized UUT or some other instrumentation or equipment is not readily available. In these cases, the test methodology might entail developing software to simulate the UUT or other parts of the system. In any event, the test methodology is outlined in the requirements specification, and details can be added as the project evolves.

At Bloomy Controls, we prioritize the requirements as critical, high, medium, and low. **Critical** items are directly related to the primary objective of the project. They are showstoppers. If any critical-priority requirements cannot be completed, we cease to proceed with the project as defined. Items of **high** priority are very important features, but not necessarily showstoppers. **Medium** is the most common priority. Items of **low** priority are features that we incorporate if time and budget permit, after all higher-priority items have been completed. Priorities are referenced during the design and implementation phases to determine the order in which the requirements are addressed and the time that we budget for each.

The requirements specification should be reviewed by multiple people involved in developing, using, and maintaining the system, if applicable. It is common to discuss, revise, and improve the specification before proceeding. A word processor with comment-insertion and change-tracking features is simple but nonetheless useful for incorporating comments and changes from multiple reviewers. More specialized requirements-management tools such as Telelogic DOORs and IBM's Rational RequisitePro provide clear visibility, change management, and traceability. Additionally, NI Requirements Gateway is an add-on package that provides a traceability link between requirements specifications formed with any of the previously mentioned tools and your LabVIEW source code. This enables you to analyze and report the compliance of your software to the project requirements.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

2.2 Design

Software is designed using traditional techniques such as flow charts and pseudo code, or using Unified Modeling Language (UML), including class, sequence, collaboration, state, and component diagrams. Many excellent references describe these standard design techniques;[3] I do not describe them here. Instead, I would like to note that Theorem 2.1 can be extended to include design documentation. The more overall planning there is, including design documents as well as specifications, the better the LabVIEW style will be.

Many developers are tempted to skip straight from the requirements specification to the software development. If fewer than 25% of LabVIEW developers write specifications, far fewer are apt to create design documents. Again, I have observed a discrepancy between hardware- and software-development practices. For hardware, the design phase is a given. Schematics always precede the soldering of circuits. Block diagrams and CAD drawings precede the fabrication of mechanical assemblies, fixtures, and equipment racks. Design documentation is essential for hardware. Why should software be any different?

A few explanations, albeit excuses, might exist. In some applications, the software design is discovered by experimenting with an implementation. For example, one or more prototype LabVIEW VIs are developed to help characterize the application that the software is intended to solve. Specifically, prototypes are often developed to determine how fast a data acquisition and online analysis routine can run on a given computer. The working prototype VIs might eventually evolve into an integral portion of the software. Many LabVIEW developers prefer to expand the prototype into the finished application, following the Code and Fix development model. This can lead to trouble, as you saw in the Spaghetti VI example in Chapter 1. Discipline is required to stop with a working prototype and to go back and complete the specification and design phases before continuing with software development.

Another explanation is that a design phase is not necessary for very simple applications. Indeed, LabVIEW applications range in complexity from very simple to very complex. If a given application can be solved by configuring a few Express VIs or examples, the design phase is not necessary. Also, if you can quickly implement an application using a combination of your favorite design pattern templates, drivers, and utilities—all of which you are intimately familiar with—and the application is strictly for your own use, do not bother creating a design document. At what point does the application's complexity become worthy of a separate design phase? The requirements specification should provide some insight—assuming that you wrote one.

Finally, sometimes a conventional design does not translate well to a LabVIEW diagram or might even adversely affect your application's style. For example, the literal translation of a flow chart to a LabVIEW diagram might entail networks of Sequence and Case structures that do not constitute good programming style. In fact, the result might appear like a cross between Nested VI and Spaghetti VI from the examples in Chapter 1, while occupying the space of Meticulous VI. This would be the worst of all worlds! Instead, it is best to understand the precepts of good LabVIEW style and to apply modeling and design techniques in a manner to complement your diagrams. For example, the State Machine Diagram specifies a design that gracefully translates into the most popular LabVIEW design patterns.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

2.2.1 Search for Useful Resources

Search for resources that will expedite design and development while positively influencing style. Search online for products, references, examples, drivers, toolkits, colleagues, and consultants. Start with `www.ni.com`. **NI Developer Zone,** in particular, is a forum for developers in need of LabVIEW resources, such as articles, example code, and support. The **LabVIEW Tools Network** is an NI-sponsored site within the NI Developer Zone that lists add-on products, books, tutorials, and more. Additionally, visit OpenG.org, and the LAVA user group. **OpenG** is an organized community committed to the development and use of open source LabVIEW tools; the URL is `www.openg.org`. The LAVA website is dedicated to the open and unbiased exchange of ideas on intermediate to advanced topics in LabVIEW; the URL is `www.lavausergroup.org`.

Do not overlook the many offline resources and media available as well. The LabVIEW environment contains hundreds of example VIs that are searchable using NI Example Finder (**Help**»**Find Examples**). The *LabVIEW Help* contains application notes and white papers. You can purchase many LabVIEW books from your favorite bookstore or online reseller. Additionally, NI offers hands-on instructor-led training through NI Certified Training Centers, and customized solutions through the NI Alliance Program. The NI Alliance program is a worldwide network of more than 600 consultants, systems integrators, developers, channel partners, and industry experts. For example, Bloomy Controls is a Select Alliance Partner and hosts two NI Certified Training Centers.[4]

Now back to the low-emissions exhaust system application example. Our requirements include control of several specialized gas-analyzing instruments and SPC analysis of the data. Hypothetically, we looked on NI Developer Zone and did not locate drivers for our specific instruments. We also inquired with the instrument manufacturers but came up empty. We did find the SPC Toolkit for LabVIEW, part of the LabVIEW Enterprise Connectivity Toolset from National Instruments. It contains VIs and charts for all the SPC analysis that we need. Hence, we can elect to use the SPC Toolkit and save ourselves substantial development effort. Also, our LabVIEW style is positively influenced by incorporating the high-quality VIs that are provided with an NI toolkit, and reducing the scope and complexity of the source code.

2.2.2 Develop a Proof of Concept

In many cases, it is useful to develop a proof of concept to evaluate a specific instrument, hardware component, or software design. For example, you might have a critical requirement for high-speed acquisition and online data processing. You might be uncertain that a given computer, instrument or DAQ module, or communications bus, combined with a data processing algorithm, can execute within a specific time interval. Therefore, you should design a test that will perform or simulate the critical acquisition and analysis algorithms, and benchmark the execution speed. You then could use the results of your benchmark tests to validate your design; seek alternatives, such as higher-performing instrumentation or faster data processing algorithms; or relax the requirements in your specification, if necessary.

In regard to the exhaust system, we need to be able to send set points to mass flow controllers, acquire data from several gas analyzers synchronized within 10ms of each other, and perform SPC analysis algorithms and chart updates in 100ms intervals. These are critical requirements. There is no use working on any of the other features until we are certain that these requirements can be satisfied. Therefore, we develop a proof of concept consisting of VIs that perform these functions on a small scale, and we set up some experiments to monitor the performance.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

When the proof of concept is complete, make sure you save it. You might be able to reuse all or portions of it in the final application. Alternatively, you might find that the performance of your completed application differs from the proof of concept. In this case, you should return to your proof of concept as a troubleshooting tool. If it still functions properly, you can begin to isolate performance issues in other areas. If the proof of concept suddenly stops working, the hardware, system configuration or other conditions might have changed since the last successful run. Reverting to a previously tested software and hardware configuration is sometimes referred to as a **sanity check**. If nothing has changed but the system no longer works properly, perhaps the only reasonable explanation left is that we, the developers, have lost our sanity.

As you might have guessed, the proof of concept positively influences LabVIEW style. Because the proof of concept normally addresses the most critical requirements of the application, the software will likely either incorporate or be modeled after the proof of concept. Implementing a routine that has already been tested and debugged reduces the risk of significant changes. Additionally, the proof of concept might help define some of the application's primary constructs, such as the design pattern and data structures. When these elements are proven, the corresponding application requires fewer changes, and fewer changes equates to neater code, per Theorem 2.2.

Rule 2.3 Maintain good LabVIEW style throughout the proof of concepts

My personal pet peeve is when developers are sloppy with their proof of concepts. For example, we have all met developers who prototype VIs using default control labels such as **numeric** and **numeric 2,** and VI names such as **Untitled 1.** They expect to show us their prototype and have us understand what it does, as if we have some type of telepathy. As discussed, a good proof of concept should be archived and should function as a project reference. This makes sense only if it is developed using good style. Ideally, developers should apply the same good style conventions on the proof of concepts that they apply to the finished application. Indeed, good style allows the proof of concept to scale gracefully into an application. Good style is important all the time.

2.2.3 Revise the Specification

When the design phase is complete and the critical requirements have been evaluated via proof of concept, revisit the specification. Changes are inevitable. Conflicts might exist between several of the requirements, and tradeoffs and adjustments might be necessary. The specification is a living document. Be sure to revisit the specification on a periodic basis and make revisions. Remove any obsolete requirements. Add any new requirements. Evaluate the impact on schedule. Elaborate on any aspects of the application that are better understood than when the project began. Maintain the specification in a controlled repository where your project team members can access it easily. Manage changes and revisions according to your organization's applicable standards.

Considering our exhaust system, we might need to update the acquisition and analysis requirements based on our finding with the resource search and proof of concept development phases. Specifically, expand the SPC requirements to include more than simple Xbar & R charts, due to the VIs available in the SPC Toolkit. Also update the acquisition and data processing rates based on the benchmarks acquired using the proof of concept. Finally, we might reduce the risk level and time budgeted for each task as a result of these shortcuts.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

2.3 Configure the LabVIEW Environment

Developers, do you

- Begin most applications with a blank VI?
- Use LabVIEW's default settings for the size, color, and fonts of GUI controls and indicators?
- Create user interfaces that are functional but bland?

Alternatively, do you spend excessive time

- Reconstructing your favorite design patterns?
- Searching previous projects for reusable source code?
- Deselecting **View As Icon** or disabling auto **wire routing**?
- Making similar repetitive edits throughout most applications?

If you answered "yes" to any of these questions, you need to customize the LabVIEW environment. If you start with blank VIs and default control properties, you might have decided that attractive panels and ease of use are not important for your particular application, or you are not artistically inclined to design a better scheme. More likely, you have discovered and applied good combinations of colors, fonts, and properties that go very well together. In this case, you might be starting with the defaults because you do not have a good system of tracking and reusing elements of your previous applications. Your favorite design patterns and user interface themes should be readily accessible as templates. The palette views and front panel and block diagram options should be customized to your personal preferences. This section covers how to configure the LabVIEW environment to save time and improve your programming style.

2.3.1 LabVIEW Options Dialog Box

The LabVIEW Options dialog box, accessed from the menu bar by selecting **Tools»Options,** is used to set preferences related to the work environment. Many of these options are related to style. For example, some block diagram options I avoid include **Show dots at wire junctions, Show subVI names when dropped,** and **Place front panel terminals as icons.** Diagrams tend to look neater without the dots, labels, and icons. Dots at wire junctions help developers distinguish between the junction of multiple wire segments and overlapping wires. However, many LabVIEW developers search for dots that represent coercions, which have negative connotations. Triple-clicking any wire can reveal its branches, and dots at junctions can be avoided. Terminals as icons and subVI labels are simply a waste of precious real estate. I recommend disabling these options. Figure 2-2 shows my own personal preferences for the block diagram options.

On the front panel, I prefer most of the default options. Seasoned LabVIEW developers really appreciate transparent labels and modern style controls. In previous versions, many of us repetitively colored our labels transparent on each and every control of every panel we created. Transparent labels look much nicer than the raised labels of the past. Modern style controls, also known as 3D controls, were one of the great innovations of LabVIEW 6, enabling our VI front panels to resemble (or transcend) their physical counterparts: the panels of traditional boxed instruments. Also related to

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

front panel style is the Alignment Grid option: **Show front panel grid** and other associated properties**.** The grid appears while in edit mode and aids the developer in visually aligning and spacing front panel objects.

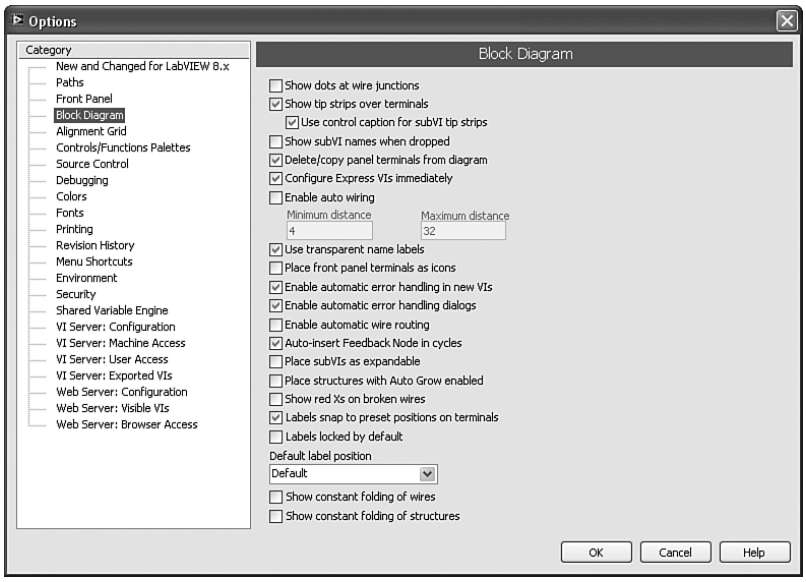


Figure 2-2
LabVIEW options dialog box, with block diagram preferences shown

Rule 2.4 Document your LabVIEW options and back up the LabVIEW.ini file

In an ideal world, all developers within an organization would standardize on the same LabVIEW environment settings. In addition to promoting consistent style, this improves the interchangeability of development machines and is a good CM practice. The LabVIEW options are maintained in the `LabVIEW.ini` configuration file located in the root of the LabVIEW installation directory. I recommend backing up this file or perhaps even saving screen shots of the options dialog box. This enables you to reconfigure a new or existing machine quickly. It is also possible to distribute the `LabVIEW.ini` file to maintain consistent options across multiple machines. However, beware that the contents of the `LabVIEW.ini` file are version, platform, and system sensitive. In addition to environment preferences, the `LabVIEW.ini` file maintains paths to the most recently used files and directories, which are relevant only to a specific machine.

Most LabVIEW developers reorganize their palettes frequently, to suit their particular editing needs at any given time. This consists of arranging the top-to-bottom order of the categories, as well as expanding, minimizing, showing, and hiding them. Additionally, custom palettes are configured using **Tools»Advanced»Edit Palette Set**. Custom palettes are strategically important with respect to code reuse.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

2.3.2 Code Reuse

Code reuse is an essential technique in all modern programming environments. The LabVIEW VI is a self-contained software module that can be called from higher-level VIs as a statically linked subVI, or dynamically by configuring **Call Setup**, or using the VI Server Call by Reference Node. SubVIs lend themselves very well to code reuse. Additionally, controls can be customized and saved as custom controls or type definitions. Design patterns and application frameworks can be saved as VI or project templates. These tools can save the developer days of development time versus starting with blank VIs and developing each application from scratch. Every developer should maintain a software reuse library containing useful LabVIEW code. Moreover, every organization that employs multiple developers should maintain a reuse library in a networked repository maintained under source control. The potential benefits include faster development time, greater commonality, and higher-quality applications.

We *all* want to save time! The time savings element of reusing source code is very important. Why reinvent the wheel if we do not have to? Always look for existing design patterns, VIs, and controls that you can reuse in your applications. Many developers use NI Example Finder to locate useful examples that are bundled with LabVIEW. This is a very well-integrated tool that presents examples in an organized and searchable manner. Far fewer developers integrate and use their own LabVIEW source code, perhaps because it might not be obvious how this is best accomplished.

We begin with the most commonly shared LabVIEW construct: the instrument driver. Although thousands of instrument drivers are available from NI Developer Zone's Instrument Driver Network (`www.ni.com/idnet`), there are tens of thousands of programmable instruments out there. Most of us have had to develop a few drivers on our own, as well as customize some of the downloaded drivers. Many organizations maintain their instrument drivers in a repository. The key to using them is to make them directly accessible on the LabVIEW **Functions** palette of the developer's computer. This is accomplished by placing copies of the desired drivers into the `LabVIEW\instr.lib` folder on the developer's machine. LabVIEW reads the contents of this folder each time it launches and populates the **Instrument Drivers** palette with subpalettes for each driver it finds. This feature facilitates seamless integration of instrument drivers, copied from anywhere, with the LabVIEW environment. Figure 2-3 shows a **Functions** palette with the **Instrument Drivers** subpalette expanded.

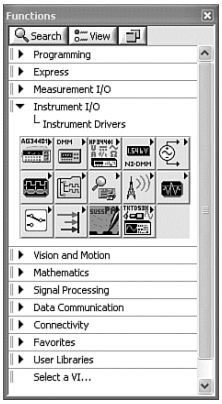


Figure 2-3
Functions palette with the **Instrument Drivers** subpalette expanded. Each icon corresponds to an instrument driver contained within the `LabVIEW\instr.lib` folder.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

Rule 2.5 Develop reusable SubVIs

- Perform specific, well-defined tasks
- Use controls instead of constants
- Apply good style

Likewise, most of us have developed subVIs for manipulating strings and arrays, reading and writing files, performing computations, and more. Any such subVIs are reusable if they satisfy certain criteria. First, they should perform a specific useful task in a manner that is flexible and general purpose. In some cases, this involves using controls assigned to connector terminals instead of constants on the diagram. Second, the subVI must be developed employing good style, per the recommendations in this book.

For example, suppose we are developing an application in which we need a routine to parse multiple data packets returned from an instrument that are delimited by carriage return and line feed character combinations. We can write a subVI that calls Match Pattern within a While Loop. If my regular expression is a string formed by concatenating the carriage return and line feed constants on the diagram, the subVI will not be very general purpose and reusable. Instead, I should create a string control on the front panel, label it **Search String**, and assign it to a prominent connector terminal. Now the subVI performs a useful and specific task: It parses data packets with variable delimiter and is general purpose and reusable.

Rule 2.6 Make reusable libraries accessible from the LabVIEW palettes

- Place reusable libraries in `LabVIEW\user.lib`
- Customize the LabVIEW **Functions** and **Controls** palettes

The final step in making the subVIs reusable is to make them readily accessible to the developer. Searching through archives that contain source code from previous projects can be cumbersome and unwieldy. Instead, identify reusable components as you code and place them in a dedicated repository. Then you can upload copies of the reusable components to locations in your LabVIEW installation directory that will integrate with the LabVIEW environment. Specifically, the `LabVIEW\user.lib` folder behaves similarly to `LabVIEW\instr.lib`. LabVIEW maps any folders, project libraries, or LLBs that it finds in this directory to subpalettes underneath the **User Libraries Functions** palette.

Figure 2-4 illustrates an example of a **Functions** palette with the **User Libraries** category expanded. In this case, the `user.lib` folder contains five libraries: OpenG, Bloomy Utilities, Bloomy Library, Win Utilities, and an unidentified library containing a generic icon. LabVIEW automatically assigns the generic icon, shown on the far right, to any folders, project libraries, or LLBs that do not have an associated menu file. The palette properties, including the folder or library mapping, as well as the arrangement of subpalettes, menus, icons, and VIs, are specified by selecting **Tools»Advanced»Edit Palette Set**.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

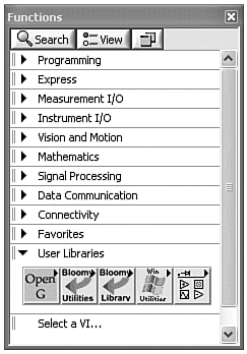


Figure 2-4
The **Functions** palette with the **User Libraries** category expanded reveals five user libraries.

SubVIs that perform low-level tasks that complement or extend the capabilities of the built-in LabVIEW functions are known as **utility VIs**. It is desirable to access the utility VIs directly from the associated LabVIEW palettes, in addition to or instead of the **User Libraries** palette. For example, the OpenG and Bloomy reuse libraries both contain File I/O libraries. These VIs are accessed by navigating the **User Libraries** palette, as shown in Figures 2-5A and 2-5B. However, it might be faster and more intuitive if these libraries were accessible from the **File I/O** palette. Fortunately, LabVIEW provides complete flexibility in customizing the palettes. In Figure 2-5C, redundant subpalettes have been created for the File I/O utility libraries on the **File I/O** palette. This is accomplished from the palette set editor by inserting new subpalettes on the associated LabVIEW palettes that are linked to the MNU files, folders, project libraries, or LLBs that comprise the utility libraries. It is important to recognize that the source files uniquely remain in the `LabVIEW\user.lib` folder. The MNU files that define the palette contents are edited by the palette set editor and maintained at the following path: `<default data directory>\<LabVIEW version number>\palettes`.

Most of us customize the controls and indicators available from LabVIEW's **Controls** palette. Some of the most common edits include changing the data type or representation, resizing, coloring the various foreground and background elements, and editing fonts of the labels or embedded text. Alternatively, the control editor can be applied to dramatically modify the appearance and create a brand new type of control. Customized controls can be saved as custom controls, type definitions, or strict type definitions, and can be reused throughout your applications. Any such controls can be accessed from the **User Controls** palette simply by placing them within the `LabVIEW\user.lib` folder. LabVIEW uses the file extensions to determine whether each item appears on the **Controls** palette or **Functions** palette. Files with the `.vi`, or `.vit` extension appear in the **Functions** palette; files with the `.ctl` extension appear in the **Controls** palette.

Templates are VIs that contain commonly used design patterns or constructs, saved with the `.vit` extension. Some common templates include the SubVI with Error Handling, Dialog Using Events, and Standard State Machine. These templates are all available from LabVIEW's **New** dialog box, which appears every time you select **VI or Project from Template** from the **Getting Started** window, or **New** from the File menu. The **New** dialog box is shown in Figure 2-6.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

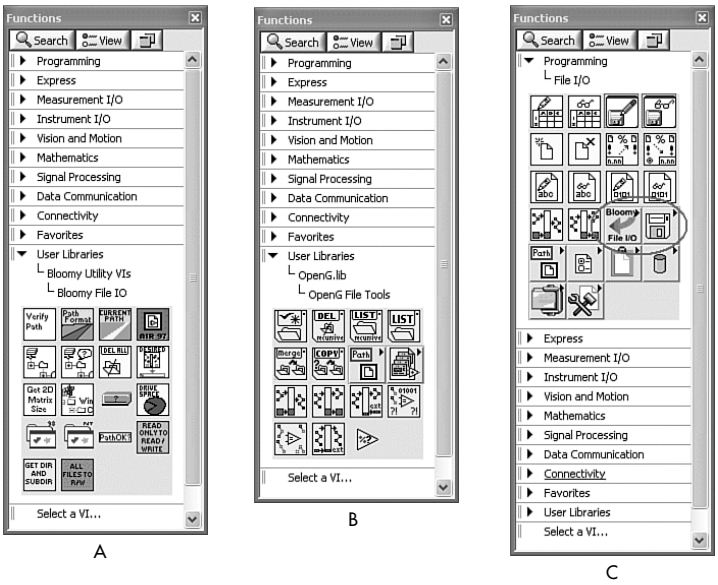


Figure 2-5
The OpenG and Bloomy libraries contain File I/O utility VIs that reside within `LabVIEW\user.lib`. The LabVIEW palettes are customized to access these VIs equally from the LabVIEW **User Libraries** palette, as shown in palettes A and B, and the **File I/O** palette, as shown in palette C.

The **SubVI with Error Handling** consists of a front panel with the **error in** and **error out** clusters assigned to the lower left and right connector terminals, and a block diagram containing a Case structure with **error in** wired to the selector terminal. The developer adds other controls and indicators to the front panel, assigns them to appropriate connector terminals, and creates and wires together diagram objects inside the **No Error** frame of the Case structure. This template might save only a couple minutes of development effort, but it can be applied repetitively, reducing tedium.

Dialog Using Events is a VI that opens its front panel when called and prompts the user with any data or selections provided on the panel. Its Window Appearance property is configured as Dialog, and it contains two dialog box–style Boolean controls labeled **OK** and **Cancel**. The diagram consists of a While Loop that contains an Event structure that sleeps until the Value Change event fires on either control, which stops the VI. If all your dialog boxes originate from this template, your application's graphical user interface will exhibit a style and behavior that is consistent with the native operating system dialog boxes and is consistent throughout your application.

The **Standard State Machine** is a common design pattern that consists of a Case structure within a While Loop, a shift register for passing the case selector value, and an enumerated type definition that contains the state names. State Machines are a popular type of design pattern that are described in much more detail in Chapter 8, "Design Patterns."

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

Rule 2.7 Place reusable templates in the `LabVIEW\templates` folder

Figure 2-6A shows the LabVIEW **New** dialog box with the aforementioned templates that ship with LabVIEW. You can expand upon the standard templates and also incorporate your own by placing them in the `LabVIEW\templates` folder. In addition to increasing productivity, templates will help improve your LabVIEW programming style, as long as your templates follow good style. Figure 2-6B shows some templates that are part of the Bloomy Controls software reuse library.

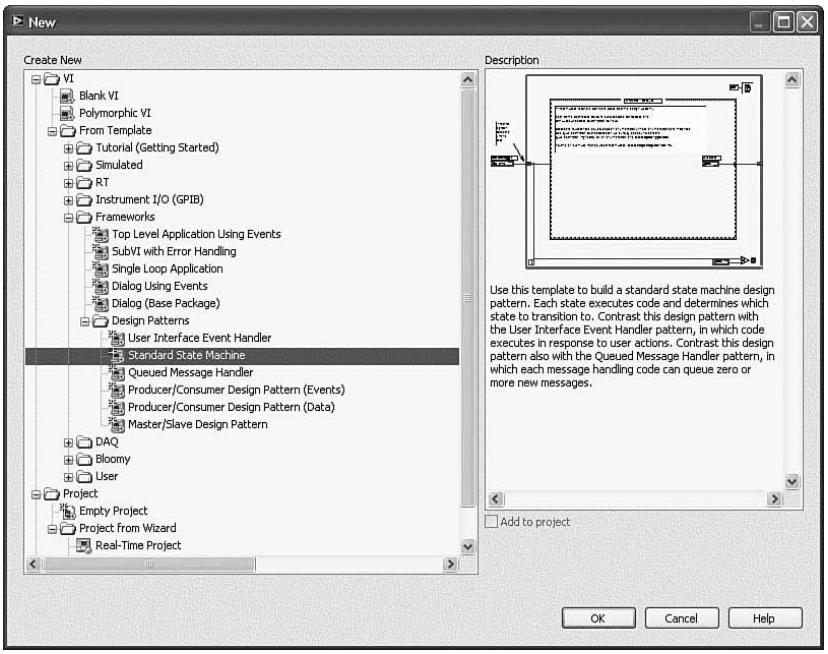


Figure 2-6A
The **New** dialog box provides access to templates that ship with LabVIEW, including the Standard State Machine.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

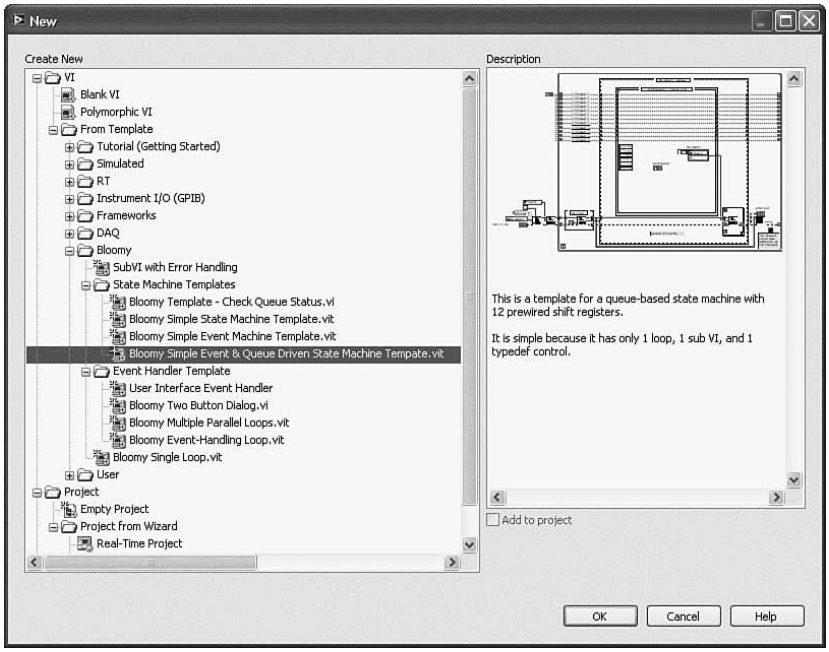


Figure 2-6B
An organization's templates are accessible from the **New** dialog box when placed in the `LabVIEW\templates` folder.

2.4 Project Organization, File Naming, and Control

LabVIEW 8.0 introduced Project Explorer, an interface for organizing project files within a project tree. The properties that specify the project are stored within an XML file called the LabVIEW project (`.lvproj`). The files referenced by the project can reside almost anywhere, can be named almost anything, and can target on a growing variety of computing devices. This presents new opportunities, as well as new challenges for developers and organizations. Conventions for organizing, naming, and controlling project files are desirable.

Many of us take it for granted that source files are organized, named, and controlled appropriately. From experience, I have learned that this assumption is entirely *not* valid. For example, it seems that some developers strive to minimize the length of their filenames. I have seen naming conventions that consist of very short acronyms followed by an underscore and number, as if there was an 8.3-character limitation reminiscent of DOS. Surprisingly, this is not uncommon. I have seen large applications that consist of hundreds of such VIs within a single folder or LLB, none of them marked as top level. In these situations, when the developer is long gone and documentation is scarce, *I wish I could buy a vowel!*

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

Furthermore, I have seen test labs that contain multiple PCs, each maintaining dozens of variations of the same application. The users edit constants on the block diagram that represent transducer scaling factors, as well as default values and ranges of controls and indicators on the front panel, and then save the VIs under a new name. Some but not all of the variations are backed up to a network server, CDs, and various media. Any significant upgrade would entail first taking inventory of every instance of the application, examining the differences, and possibly making redundant edits to multiple variations of the application. Instead, establish some conventions for file organization, naming, and control.

2.4.1 Disk Organization

Rule 2.8 Maintain an organized repository on disk

Because the LabVIEW project allows the project's files to reside almost anywhere, a project that appears well organized within the Project Explorer window might have source files scattered throughout your hard drive, network servers, peers, storage devices, and miscellaneous targets. It can be a CM nightmare if the files are accessible to other projects and users, without source control, or if any of the files are moved independent of the project. Also, it becomes difficult to manage and maintain the files using conventional operating system utilities, such as Windows Explorer, when the project associations are transparent to the operating system and the files are scattered in many places. Therefore, create an organized file repository on disk for every project, to group the project files and place them under source control. Create a folder hierarchy to maintain organization of the files within the repository. Figure 2-7A illustrates a project file repository, with separate top-level folders for `Application Build`, `Data`, `Documentation`, `Graphics`, and `LV Source`. The `Application Build` folder contains the executable and install utility. The `Data` folder contains sample data sets generated by the application for the developer's reference. The `Documentation` folder contains reference documents, such as project specifications and instrument user manuals, as well as documents that ship with the product, such as help files and user manuals. The `Graphics` folder contains image files such as screen shots, company logos, and icons. The `LV Source` folder contains LabVIEW source files that are uniquely developed for the project. Source files from a reuse library that are used by the project but not edited within the project are located in a separate repository for use in multiple projects. This includes reusable utilities and components such as instrument drivers.

Rule 2.9 Create a LabVIEW source folder hierarchy that reflects your application's architecture

The five folders shown in the `[Project Name]` folder of Figure 2-7A form a base set of file categories. If your project files are located in the roots of these five folders, all of which reside within a single project folder, you have successfully created a project repository. However, each of the folders can be further divided into subfolders for greater organization on disk. In particular, you might create a subfolder hierarchy to organize the LabVIEW source files in a manner that reflects the application's architecture. For example, some typical subfolder names include `Analysis`, `Configuration`, `DAQ`, `Data Manipulation`, `File IO`, and `User Interface`. I recommend placing the top-level VIs at the root of `LV Source` and placing all subVIs and controls in the appropriate subfolders. Likewise, the other base folders can be further organized into subfolders. Figure 2-7B provides an example of an expanded folder hierarchy for a complex application.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

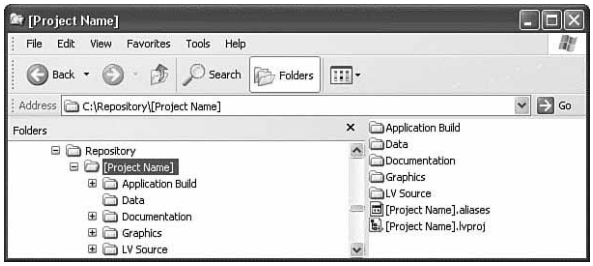


Figure 2-7A
Folder hierarchy for maintaining project files on disk, with a basic set of top-level folders

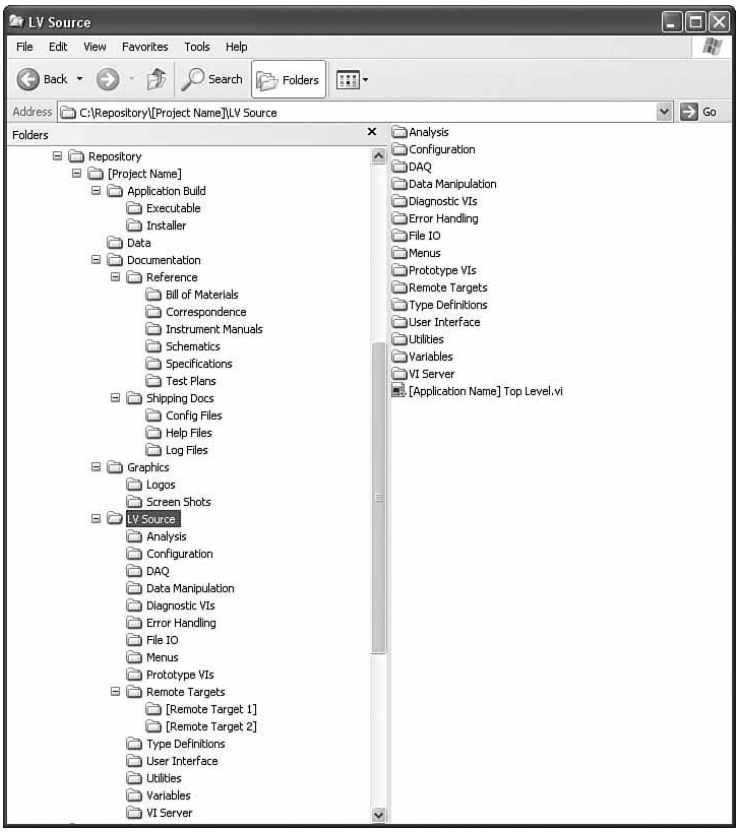


Figure 2-7B
Expanded folder hierarchy that provides detailed organization and an accelerated starting point for complex application development

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

Most developers agree with the usefulness of an organized repository. However, like many Rules, we often start out with good intentions but might lose focus under the pressures of tight deadlines. Unless we form good habits, we risk sacrificing many Rules, including the source folder hierarchy. Furthermore, it is problematic for the LabVIEW project, as well as most source code control tools, to move source files among folders on disk. When the files are referenced within the project, moving them on disk causes file-linking errors. So here is the secret to maintaining an organized repository:

Rule 2.10 Create the folder hierarchy before you begin coding

If you initiate your project with an organized folder hierarchy from the outset, maintaining an organized repository is as simple as saving each file in the appropriate folders as you create and edit the files throughout the development cycle. I find that it is much more feasible if the folder hierarchy already exists. Furthermore, you need not create a new folder hierarchy for every project from scratch. I find that most projects I develop have many of the same general categories of files, similar to Figure 2-7B. After you have created one such folder hierarchy, use it as a model to create a reusable folder hierarchy template.

Furthermore, the folder hierarchy template can contain any number of source file templates, including a LabVIEW project file, a top-level VI template, documentation templates, and more. Indeed, the folder hierarchy integrated with source file templates constitutes an accelerated starting point for application development using good style.

On a final note, it is important to realize that the folder hierarchy template is intended as a starting point, not a one-size-fits-all repository. For the folder hierarchy to reflect the application's architecture, you must customize the template for your project by adding and removing folders as necessary. To the extent possible, customize the template before you begin coding, to avoid moving source files on disk later.

2.4.2 The LabVIEW Project

After an organized repository has been established, proceed to the LabVIEW project. Use Project Explorer to organize the project files into a hierarchy of folders that reflects the application's architecture, similar to the project repository. If you began with an organized folder hierarchy on disk, it is extremely simple to create an organized LabVIEW project. In Project Explorer, start with a new LabVIEW project or template, right-click on My Computer or another target, choose **Add Folder** from the shortcut menu, navigate to the project repository, and choose an existing folder. Repeat this process for each folder in the repository that you expect to utilize during LabVIEW development. This generally includes all LabVIEW source folders appearing under `LV Source`, as well as requirements specifications, instrument manuals, and other reference materials. Figure 2-8 provides an example. Dissimilar to the repository on disk, however, you are free to move file references within Project Explorer as often and as randomly as you like. How you arrange files and folders on the project tree has no effect on the location of the corresponding files and folders in the repository.

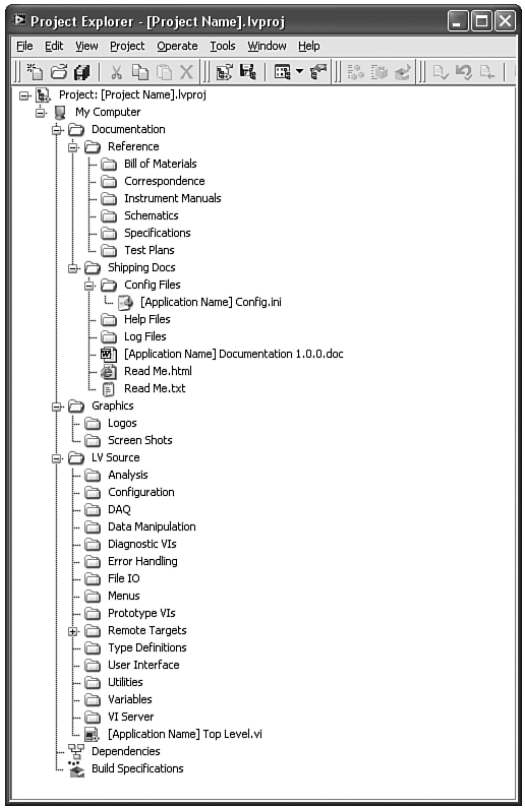


Figure 2-8
The LabVIEW project contains an organized folder hierarchy that reflects the application's architecture, similar to the project repository.

Rule 2.11 Organize LabVIEW source files into cohesive project libraries, where appropriate

The LabVIEW project library is a file with the `.lvlib` extension that maintains properties shared by a collection of LabVIEW source files. The properties include a name prefix, version, access rights, password, palette icons, and menu names. Project libraries are intended to maintain cohesive collections of source files that work together to perform a specific set of functions, such as an instrument driver. Project libraries prevent namespace conflicts that arise when two files containing the same name are opened. Specifically, all the source files that comprise the project library inherit the library name as the prefix for the source filename. This allows project libraries to contain identically named source files, as long as the library names are unique. For example, multiple instrument drivers can contain an Init VI and a Close VI, which can be loaded in memory at the same time. The file prefix is

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

provided by the `.lvlib` filename, which is unique to each instrument driver. Additionally, you can use the Library to define VIs as either public and accessible to all users or private and usable only by the developer. With the ability to control which VIs are usable, you can maintain a consistent public VI interface while changing the underlying private VIs without fear of breaking the user's code. Note that legacy instrument drivers, toolkits, and other libraries developed prior to version 8.0 are generally distributed in a compressed file format known as the LLB. LLBs are a different type of entity than a project library. LLBs physically contain the source files that they reference, whereas project libraries are XML files that contain references and property values for the files that comprise the library.

Libraries distributed as LLBs have several disadvantages compared to folders and project library files. These include lack of organization, limited compatibility with operating system utilities, longer load and save time, and risk of file corruption. Specifically, LLBs provide only two levels of organization: top level and not top level. Operating system utilities such as Windows Explorer cannot search for source files within LLBs. Source control tools cannot check the individual files in and out within an LLB. Additionally, because the source files are compressed within the LLB, they take longer to load and save. Finally, many people tend to pack too many source files into one LLB, further reducing organization and risking loss of work from file corruption.

Fortunately, LLBs can be converted to folders of VIs with project libraries. First, use the LLB Manager (**Tools»LLB Manager**) to convert the LLB to a folder of VIs. Then go into Project Explorer and add the folder of VIs to a project. This creates a project folder that contains the library VIs. Next, choose **Convert to Library** from the project folder's shortcut menu. Finally, name and save the project library.

2.4.3 File-Naming Conventions

This section presents file-naming conventions for the LabVIEW source files.

Rule 2.12 Create unique and intuitive source filenames

It is important to use intuitive filenames that describe their primary function. As a rule, the source file and project library names should combine to uniquely identify each file. Consider the hypothetical situation that you might need to take an unexpected leave of absence during the middle of an important project. Your colleagues should be able to locate and identify your source files, understand your programming style, and resume project development in your absence. This is a true test of good style.

Rule 2.13 Do not abbreviate filenames

Avoid conventions that rely on very short abbreviations, acronyms, and numbers. For example, the source filename `IPS_Osc_H_to_0.vi` might be meaningful to the original developer, but not to many others. Use as many characters and distinct words as necessary to uniquely identify the source file. Avoid using characters that not all file systems accept, such as slash (`/`), backslash (`\`), colon (`:`), and tilde (`~`). LLB Manager and VI Analyzer are tools that can be used to test the platform portability of your filenames.

Rule 2.14 Never use LabVIEW's default filenames

Never rely on a default name and number to uniquely identify a source file. This contradicts LabVIEW's built-in default VI naming convention, in which LabVIEW automatically assigns the

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

name **`Untitled`** *<number>* and **`Control`** *<number>* to each new VI and custom control or type definition, respectively. Avoid default names and numbers at all costs. In my opinion, this is a severe violation of good style.

Many developers append numbers to the filenames to indicate the version number. Note that this is not necessary if your source files are maintained within project libraries or if the project repository is under source control. Additionally, VI revision history (**Tools»Source Control»Show History**) is a built-in LabVIEW feature that maintains the revision number of your VIs. For example, it can be configured to automatically increment and prompt for comment each time you save or close a VI with changes. This is the only method in which the revision number is maintained directly within the source file.

Legacy LabVIEW Plug and Play instrument driver files developed prior to the LabVIEW project follow a specific file-naming convention. Each filename contains an instrument prefix, which is an abbreviation of the instrument vendor name and the instrument model. For example, the Keithley 2000 digital multimeter has the prefix **`ke2000`**. In addition, multiple required files are explicitly specified in the VXI*plug&play* Instrument Driver Functional Body specification. You can view the complete specification at `www.vxipnp.org`. However, modern project-style instrument drivers maintain the instrument prefix in the project library instead of in the source filenames. Hence, the instrument prefix is no longer needed within the individual filenames.

Rule 2.15 Identify the top-level VIs

The most important source files to identify in any LabVIEW project are the top-level VIs. Always distinguish these files by maintaining them near the root of the folder hierarchy on disk, as well as within the project, and apply an appropriate naming convention, such as *<project name>* Main VI, or *<project name>* Top Level VI. Figures 2-7B and 2-8 illustrate this convention. Because the top-level VI is not contained within any of the project's libraries, the project name is included within the filename. This ensures that the filename is unique and intuitive.

2.4.4 Source Control

Source control is a process by which source files are secured, shared, and maintained in a multideveloper environment. Source control is facilitated by a third-party source control application. LabVIEW integrates with multiple source control applications by enabling us to perform the most common source control operations from within the LabVIEW project. As of LabVIEW version 8.2, LabVIEW supports Microsoft Visual SourceSafe, Perforce, MKS Source Integrity, IBM Rational ClearCase, Serena Version Manager (PVCS), Seapine Surround SCM, Borland StarTeam, Telelogic Synergy, PushOK (CVS and SVN plug-ins), and ionForge Evolution. Under Windows, LabVIEW uses the Microsoft Source Code Control Interface. On non-Windows platforms, LabVIEW supports only Perforce via the command-line interface.

Source control is essential in a multideveloper environment. It allows a project team to collaborate on the same project at the same time by controlling access to the project folders and source files. This prevents multiple developers from editing the same source files at the same time. Also, most source control packages enable you to configure permissions for each user. For example, you could configure a project documentation repository to give write access only to managers, while you could configure your source code repository to give write access only to software developers. Alternatively, some source control tools give multiple developers unlimited access to the source files, and the source control tools resolve conflicts when changes are merged back into the repository.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27

Source control tools track changes and revisions to all types of project files and maintain a comprehensive backup. These features are as beneficial for a single-developer project as they are for a multi-developer project. Additionally, CM and source control are common requirements for different certifications, such as ISO 9000.

Rule 2.16 Follow your organization's CM Rules

If your organization has a CM process in place, follow all Rules that have been established, including source control. There could be a specific procedure that you must follow that includes specific tools and configuration settings. Placing a project under source control usually involves creating or moving your repository to a designated location, adding the files to source control, and configuring the access rights for the files within the source control software. You can then proceed with development from the LabVIEW environment, adding new files to source control as they are created, checking out existing files for editing, checking them back in, and so on. For more information, refer to the *LabVIEW Help*.

Rule 2.17 Avoid moving source files on disk

One point of emphasis is that after development has started, project files can be organized freely within Project Explorer but should *not* be moved on disk. Otherwise, broken links will result within the LabVIEW project and any source control tools that reference the dislocated files.

Now we are ready to begin coding!

Endnotes

1. Howard M. Kanare. *Writing the Laboratory Notebook*. Washington D.C.: American Chemical Society, 1985.
2. Free downloadable materials are available from `www.bloomy.com/resources`.
3. Software design references:

   Larman, Craig. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, second edition. Upper Saddle River, NJ: Prentice Hall PTR, 2001.

   McConnell, Steven. *Code Complete*, second edition. Redmond, WA: Microsoft Press, 2004.

   McConnell, Steven. *Software Project Survival Guide*. Redmond, WA: Microsoft Press, 1997.
4. Bloomy Controls is an NI Select Integration Partner that provides systems development and training services throughout the Northeast United States, including offices in Windsor, Connecticut; Milford, Massachusetts; and Fort Lee, New Jersey.

The LabVIEW Style Book By Peter A. Blume - President, Bloomy Controls, Inc. ISBN: 0131458353 Publisher: Prentice Hall
Print Publication Date: 2007/02/27