User Symposium/89

Michael Porter
Codex

LabVIEW Program Development in Large Test Networks

NATIONAL INSTRUMENTS®
The Software is the Instrument™

LabVIEW™ Program Development in Large Test Networks

By Michael L. Porter
Codex Corporation
20 Cabot Blvd.
Mansfield Ma. 02048

Abstract:

A "Test Network" is defined as any collection of hardware and software that, together, provides the user with the capability of investigating and measuring the performance of a specific test subject. There are, however, some key differences between developing a large test network and a small one. In the context of a LabVIEW™ environment, this paper will present several key software design concepts that have been found to be essential for large development efforts, but are applicable to the creation of test systems of any size:

- The difference between Testing and Experimenting will be explored. Special attention will be paid to the importance of this distinction as it applies to: (1) identifying of the functions to be performed, (2) selection of equipment to be used and (3) the overall structure of the resulting test network.

* Concrete examples will be given of how to use the principles of Information Hiding and Designing for Change to functionally decompose your problem such that the resulting test implementation is efficient, error-free and maintainable.

* The advantages of a robust Software Hierarchy will be demonstrated. A proposed model will be presented, along with specific guidelines for utilizing it as a design tool to aid the system designer in managing the development effort.

The intended audience for this paper is interdisciplinary in nature, but with some experience in testing and/or process control. However, no software design experience is assumed.

LabVIEW™ Program Development in Large Test Networks

1. Introduction

For the past year and a half I have been involved in the design and production of a generic automated test station for data communication devices. The users of this station include groups involved in the development and production of a broad range of products ranging from low-speed dial modems to premium lease line modems to high-speed CSU and ISDN devices. In addition, as an independent contractor, I have been involved in specifying and designing systems for use in process control and medical research.

In order to realize these accomplishments, I have had to orchestrate the operation of up to four Units Under Test (via a proprietary serial protocol) and test networks consisting of as many as 10 to 15 pieces of GPIB and RS232 controlled test equipment plus digital and analog data inputs and outputs. Due to the size of these programs, I have had to resolve a number of issues usually not encountered in LabVIEW program development. My intention is to cover these issues in more or less the order in which they arise in a development project.

The first step will be to look at the foundation upon which the remainder of your work will rest -- the **Test Environment**. The goal of this section is to enable you to detail <u>exactly</u> what it is that you are trying to do. This will be accomplished through an examination of the factors that determine (1) the test subject and (2) the type of test required.

Next, having defined the requirements to be satisfied, we will explore the issue of **Functional Decomposition**. Our discussion will center on how to use the principles of "Information Hiding" and "Design for Changeability" to deepen your understanding of the challenges presented by your project. Specifically we will cover program documentation, the theory of abstractions and guidelines to be used in task decomposition.

Finally, a model for the finished application will be presented in the form of a generic LabVIEW **Software Hierarchy.** Special attention will be paid to providing a functional description of each layer, as well as suggestions on making use of the model.

Before I begin though, I would like to comment on a particularly common "myth" concerning LabVIEW. You have probably heard many times, as I have, that LabVIEW allows a computer neophyte to create test applications rapidly and efficiently *without programming*. At the risk of incurring the ire of the kind folks at National Instruments, let me state that this is simply not true.

The fact of the matter is that LabVIEW is a formal programming language. While I am not questioning the fact that it's graphic programming environment allows perfectly serviceable applications to be "hacked" together rather easily, the goal of this paper is to demonstrate the benefits that can be realized by adhering to good software design and implementation practices. Moreover, I intend to show that time invested in learning these basic principles will pay the system designer significant dividends in the form of shorter development time, increased functionality and execution efficiency.

Obviously, no paper of this size could reasonably hope to cover all these issues exhaustively. Therefore, I have included in Appendix A a complete list of the primary source documents used in the preparation of this presentation.

2. Your Test Environment

2.1 The Important Questions...

It's been my experience that the best way to learn the inner workings of any process like test development, is to examine the presumptions upon which the process is based. In most cases, the most basic (and least questioned) assumptions have to do with what it is that you are trying to accomplish. With this in mind I would like to share with you an incident that occurred when I was a young boy growing up in southern Missouri. This story will serve as a metaphor for our first discussion.

It seems that near my home there was a local man who earned a living by carving wooden Indians -- like those which stood in front of cigar stores at the turn of the century. I remember one time, the local television station decided there was a story in this, and so dispatched a reporter to interview the wood carver. After all the usual questions about how long he had been carving wooden figures and the tools he used, the young reporter asked the man how does one go about creating a wooden Indian. "Well," the man said "it's simple, you just lay out the log and cut off anything that don't look like an Indian."

This wood carver is not unlike from us. He had to confront a block of raw potential (the log) and decide how it was to be used. Did he want to create a wooden Indian or a coffee table or maybe a bonfire? It would have served any of those uses equally well, but before he could start he had to decide. In the same way, we have to take the capabilities provided by the hardware and software at our disposal and "carve" out of them a system which will meet our users requirements. First though, we have to identify what those needs truly are. The single biggest time-burner on any development project is starting work before you know what you're trying to accomplish. You invariably end up incorporating into your system things that add nothing to the finished product but complexity.

In the context of Automated Testing, this issue can be summed up by the following questions, which we will be examining in detail in the next two sections:

1. What am I testing?
2. What kind of a test is it?

2.2 What am I testing?

All tests, by definition, have a subject -- a thing to be tested. Traditionally, it's been assumed that the subject of a test is an object, like a modem or box of laundry detergent. In contrast to that view, I assert that there are three distinct classes into which a test subject might fall:

The first class of test subjects, concerns the situation in which you are testing a product. An example of this would tests performed by an engineer working in a Product Development group. About the only thing the designer of a new product can safely assume is that Ohm's Law hasn't changed. In a new design, you could discover anything from a component with marginal performance for the application, to a bug in the software.

The second class is represented by a tester in a manufacturing organization. In this case, much more can be taken for granted. He or she can, for instance, assume that the various hardware components are being used in an appropriate manner. Likewise, software is generally not an issue since bugs were corrected much earlier in the product development cycle. In short, the design delivered by the Product Development group is assumed to be error-free. All that's left is for manufacturing to implement it in an error-free manner. In this case, the product's only purpose in the test system is to serve as a carrier of information about the manufacturing process. The true subject of the test, therefore, is not the product, but the process.

The third possibility is faced by an engineer in an Advanced Research and Development group perfecting a new modulation mode for a modem. Here, the subject of the test can't be a particular product or process, since neither exist. Rather, the researcher is developing the basic knowledge which will allow them to be created. In this case, the <u>hypothesis</u> the researcher has concerning the potential operation of the modem is what's being tested.

To summarize then the two main points: (1) The subject of a test doesn't need to be a physical entity, as it was in the first case. (2) The thing (object) that is being tested isn't necessarily the subject of the test.

2.3 What kind of test is it?

After we have defined what we are testing, the next thing to consider is what kind of test is required. This is important because it gives you your first look at the real size of your application.

It can also give you a sense of the hardware, software and intellectual resources you will need to successfully complete the project. There are two basic types of tests that can be performed:

Validation Test

A Validation Test consists of testing the performance of the subject against an external, possibly arbitrary, standard. These directives may be industrial, governmental or corporate in nature. In the case of my industry (Data Communications), this would include standards issued by the Federal Communications Commission, AT&T, CCITT and Underwriters Laboratories. An important point is that these standards not only define the required performance of the subject, but also limits the scope of the testing to be performed.

The primary attribute of a Validation Test Station is stability. What's needed is a system capable of rigorously verifying every specified attribute of the product under any and all conditions, in the exact same way, as many time as is necessary.

Experimental Test

Unlike a Validation Test, the Experimental Test has no external standards to define (and limit) it's range of applicability. In some sense, an experiment is never "finished", but is constantly evolving to conform itself to the changing need of the user. When working in this type of environment, the question that has to be in the front of the experimenters mind is, "Why?", closely followed by, "What if?".

Because of this, the primary attribute of an Experimental Test Station is flexibility. Carried to it's logical conclusion, the capacity to dynamically reconfigure itself would be a reasonable goal for such a station.

From this it can be seen that, given the same test subject, the choice of either a Validation or Experimental Test Station would result in significantly different test networks. Both tests do have one thing in common, however, the absolute requirement for accurate, repeatable results.

It should also be obvious that the definition of any particular test situation as either "Validation" or "Experimentation" is highly fluid. As an example, consider the product designer cited in our earlier discussion. At some point, the Unit Under Test (UUT) will most likely fail a test. As soon as it does, the interest of the tester shifts from whether or not it conforms to a particular specification, to the question of why it failed. This subtle change in focus takes the user of the station into an exploration of the internal workings of the subject and out of the scope of the specifications with which we are attempting to conform. This testing therefore, is experimentation -- not validation.

In the same way, experiments can slowly evolve into validation tests as the concept being tested moves from Advanced R&D to embodiment in a product.

3. Functional Decomposition

3.1 Components

Over the years, many excellent papers have been presented on the subject of "Structured Programming". The primary tenant of this approach is the requirement to construct your program out of small, carefully defined modules. As we shall see, a major advantage a LabVIEW developer has in implementing software in accordance with this criteria, is that as a language, LabVIEW incorporates all of the advantages of structured programming, but with few of the limitations. Through it's method of iconic representation of functions, it provides a nearly ideal environment for structured code development. In the end though, it is only a tool, and tools can be dulled or misused. Therefore, one of the key distinctions I have developed is that of the "component". As I use the term, it can be defined as:

> ... Any group of data structures and/or algorithms which share information on how they operate and are contained in a single icon.

Please note that there are a few differences between a component and the concept of "module" used in other programming languages. In the first place, a module usually refers to a software entity of some type, such as a data stack or a block of executable code. A component however, can be either hardware or software because LabVIEW provides an environment in which any kind of system resource can be condensed to an icon on the screen. In this way, the distinction between hardware and software becomes truly irrelevant since the interface and the operation are the same for either. This process is called "abstraction" and is the means by which the essential descriptive properties of a function are extracted such a way that the resulting functional description can be used in place of the function itself.

Secondly, unlike modules, components are always standalone programs. Testing a component, therefore, does not require the creation of short test programs or "drivers". Debugging consists of simply entering the appropriate values into the front-panel controls and executing the program. (When creating a large test system, this can be a significant advantage since the generation of separate test programs can, typically, double your development effort.) This in turn, results in test programs that are more thoroughly tested and, therefore, less likely to spring unpleasant "surprises" on the user.

Finally, LabVIEW components are assumed to exist in a "data flow" environment and so are seen as essentially static objects transforming the data rather than complex conditional operators making decisions based on it.

3.2 Criteria for Decomposition

In an earlier discussion, we saw that a key attribute of a component was information sharing. The corollary to this, is that information hiding is a key criteria for Functional Decomposition. To see how this works, let's start by considering the two basic types of information contained in any system.

The first type consists of algorithms which once written, are never changed. This is called **logical information.** An example of this type of information would be an algorithm for performing a numerical calculation such as the derivation of a square-root.

The second type consists of things which might change at any time for purely arbitrary reason. This is called **factual information.** It includes things like the type of product you are testing or the maximum permissible value for a given parameter.

The goal of this type of decomposition is to divide the problem using information hiding as the primary criteria so that any changes to the factual information contained in a given function, are invisible from outside that function. In particular, no consideration should be given to the probable "flow of execution" of the resulting program. In a 1972 paper [3], Parnas summed it up this way:

> "... it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions that are likely to change."

(As an aside, managers involved in implementing this method for the first time should recognize that this may call for a significant mental shift on the part of some designers.)

An example of how this principle could be applied to a real-world situation can be seen in the Carrier Detect Threshold test I wrote as part of my first test station. The algorithm I used was as follows:

1. Configure the modems for the appropriate Threshold Setting.
2. Set the receive level of the UUT to 2 dB greater than the "Off Threshold".
3. See if the Carrier Detect signal on the UUT is still active.
4. If it is, decrease the UUT receive level by 0.2 dB and go back to step 3.
5. Measure and record the current UUT receive level as the "Off Threshold".
6. Increase the UUT receive level by 0.2 dB.
7. See if the Carrier Detect signal on the UUT is still inactive.
8. If it is, go back to step 6.
9. Measure and record the current UUT receive level as the "On Threshold".
10. Repeat steps 2 thru 9 for all remaining Carrier Detect Threshold setting.

While each step would be impacted by the design philosophy we are discussing, the portion of the process I wish to concentrate on is the measurement of the UUT receive level (steps 5 and 9). Using the terminology given above, there are at least 4 pieces of **factual information** in this one sub-process:

1. The type of meter used.
2. The signal being measured.
3. The dynamic range of the measurement.
4. The threshold being measured.

As we shall see, it was fortunate that I took these thing into consideration when decomposing that portion of the test.

My first approach, was to connect a GPIB controllable Digital Volt Meter (DVM) to the receive terminals, make the measurement and then convert the reading from volts to decibels. After completing the first series of tests, however, I discovered that the results were wildly inaccurate. The problem was eventually traced to the DVM. I had selected a one capable of not only taking the measurement but performing the math for the data conversion as well. The trouble was that it had a bandwidth of DC to 250 Khz while the signal I was measuring had a bandwidth of 500 Hz to 3400 Hz. The end result was that spurious pickup of out-of-band energy caused an apparent "noise floor" 30 to 40 dB higher than what actually existed.

The solution was to change the instrument I was using to monitor the signal. Instead of the DVM, I substituted a Dynamic Signal Analyzer (DSA). This instrument had the ability to read the power density between any two frequency points with an accuracy of a few tenths of a hertz. Normally, this profound a modification in test methodology would have required significant reworking of the entire test, but because I decomposed the problem recognizing that the instruments used in the testing might change, the modifications were limited to one Virtual Instrument (VI) called -- amazingly enough -- "Read Receive Level". From the stand point of the Carrier Detect Threshold test, it would be fair to say that a Dynamic Signal Analyzer didn't exist. Through the use of abstraction, I was able to extract the "essential descriptive properties" of the required function and then use that information to create a version of the function that used the DSA to make the actual measurement. In the end the DSA lost it's identity as a Dynamic Signal Analyzer and became a simple decibel meter.

(As a side note, in the latest version of our test station we have switched from the DSA to a Transmission Impairment Measurement System or TIMS for taking the reading. Moreover, the final output is now the average of 10 samples, but as in the first case, the only software impact was the generation of a new version of "Read Receive Level".)

3.3 Decomposition Guidelines

The following guidelines [1] together form the bedrock upon which any project can be built:

1. *Select a project as advanced as you can conceive, as ambitious as you can justify, in the hope that routine work can be kept to a minimum; hold out against all pressure to incorporate into the system expansions that would only result in a quantitative increase of the total amount of work to be done.*

2. *Select a machine with sound basic characteristics; from then on try to keep the specific properties of the configuration for which you are preparing the system out of your considerations as long as possible.*

3. *Be aware of the fact that experience does no means automatically lead to wisdom and understanding; in other words, make a conscious effort to learn as much as possible from your past experiences.*

The first item is concerned with the attitude of the developer as he or she approaches the Customer Statement of Work or Functional Requirement. It speaks to a kind of inner passion that drives the system developer to produce not only *a* solution to the problem, but *the best* solution. It speaks to a kind of personal integrity that refuses to allow "good enough" to be good enough. Finally, it recognizes that regardless of what may or may not be on paper, this attitude is the basic contract between you and your client.

At this point, it's appropriate to discuss the two basic kinds of problems from which a Functional Requirement might suffer -- under-specification and over-complication. In my experience, you as the system designer, have a great deal of control over both of these situations since the most common cause for either is the customer making inaccurate assumptions concerning your work. In the case of under-specification, the scenario would go something like this:

You have worked for weeks on a project and are well ahead of schedule, so you decide to show it to the client for the first time. As you run your application through it's paces, you know they're is going to be impressed by what you have accomplished. However, instead of getting the kudos you expected, the response is something along the lines of: "That's really nice, but there are a few things I was wondering if you could add to it...". That's when you realize that the Functional Specification you've been working from wasn't based on the customer's <u>needs</u>, but on their <u>perception</u> of what you can accomplish.

In the same way, if you consider over-complication, you will find a similar hidden agenda. More that likely your client is trying to protect themself from a reoccurrence of a bad past experience. Maybe they had been burnt by an unscrupulous consultant before, or maybe the problem is a basic mistrust of test engineers ("All they do is find problems with a product I'm busting my butt to get working!"). The solution to this problem is to be found in developing a sense of partnership with your customer. Be prepared to do what ever it takes to insure that your professional relationship serves the needs of both you and your customer. Don't depend on "sorting it out later" -- handle it now.

Next, we turn to an area where we are caught in a double bind, of sorts. Despite what most programmers think, no piece of software exists in a vacuum. At some point, real object code must be executed on real hardware (CPUs, Test Instruments, etc.). The difficulty is that while we need to know what the hardware is, we want to limit our dependence upon any particular device. The solution is however simple -- never forget that any piece of hardware used in the process is probably the most basic piece of factual information in the system, and treat it accordingly.

The third principle takes the form of a reminder. Wisdom and understanding aren't synonymous with advanced age or experience, rather they come about over the course of a life spent in on-going study with a conscious commitment to precision and rigor. In short -- stay awake.

3.4 Goals of Decomposition

The previous example is valuable because it not only illustrates the concept of Information Hiding, but also highlights a number of the benefits of this type of decomposition:

1. **Language Extension** - Because LabVIEW makes no distinction between it's own built-in functions and the ones you create, programming can be seen as a process of developing extensions to the language (custom icons) suitable for satisfying your specific need.

2. **Looser Coupling** - Since the internal operation of an icon is immaterial to the program using it, it is less likely that an internal change will effect other portions of the code.

3. **Simplified Debugging** - Functionality is isolated into separate routines where they can be exhaustively tested.

4. **Robustness** - Unavailable pieces of instrumentation can be simulated by appropriate software routines, thereby allowing code development to continue in spite of hardware failure or differences between the code development and test execution environments. (In the case of my work at Codex, most of the test software is developed on a Mac SE with no GPIB interface and then transferred to a Mac II for execution.)

   In the same way, totally different types of instruments can be used to provide the same functionality. In an environment where downtime can not be tolerated (such as manufacturing), this can reduce your dependence upon unique, single sourced devices.

5. **Flexibility** - Significant changes in the functional requirements of the station can be absorbed with little or no software impact. (My current Bit Error Rate test

has been used to test the performance of modems, low speed secondary channel data and is currently being used in the development of Digital Signal Processing code -- all without any modification what so ever!)

6. **Standardization of User Interfaces** - One of the facts of life in the world of automated testing is that there is no such thing as an "ideal" instrument. Any device is both defined and limited by the design decisions made during it's development. Unfortunately for those of us who make a living out of automating things, the area of remote control often suffers from design decisions that were intended to optimize the manual control interface. You will find things like half-duplex serial control ports that will only accept 1 character at a time or GPIB instruments that clear remote control errors by resetting the entire box.

   Functional Decomposition allows the test designer to hide these types of instrument problems and superimpose on the device an alternate user interface of his or her choosing.

To this point, we have been dealing with the process of decomposition itself, but little of this will be of any use to us if the results of all our careful design work aren't communicated effectively.

3.5 Documentation

In ancient times the Romans worshipped a god named Janus. This minor and rather obscure deity, had two faces, one facing forward in time, the other back. In much the same way, a program design document also has two distinct phases. To the Software Developer, it serves as a map outlining the path to the goal. To the System User/Modifier, it is a record of how the developers got where they were going and the choices they made along the way.

Having said that however, some caution is in order. This task, poorly handled, can dampen the enthusiasm of even the most stalwart team member and stem the creative efforts of the entire organization.

In order to avoid the two extremes of chaos, on the one hand, and a "paper straight-jacket", on the other, it's necessary to have some direction on what should be included and how it's to be documented. A 1972 paper [2] which addressed this entire issue of software specification, made the following points:

First, in order to reduce the likelihood of misunderstanding, the language or other medium used for communicating the specification, should be in common use by both the user and implementer. In addition, it should be sufficiently formal such that the module descriptions could conceivably be machine tested for consistency, completeness and accuracy.

Second, the actual information content needs to be controlled much as it will be in the program itself. Specifically, the intended user should be provided with all the information needed to use the function correctly, <u>and nothing more</u>. Likewise, the specification should supply the implementer with <u>only</u> the information he needs to complete the program, and no additional information; in particular, nothing should be conveyed about how the program will be used.

4. Robust Software Hierarchy

4.1 The Model

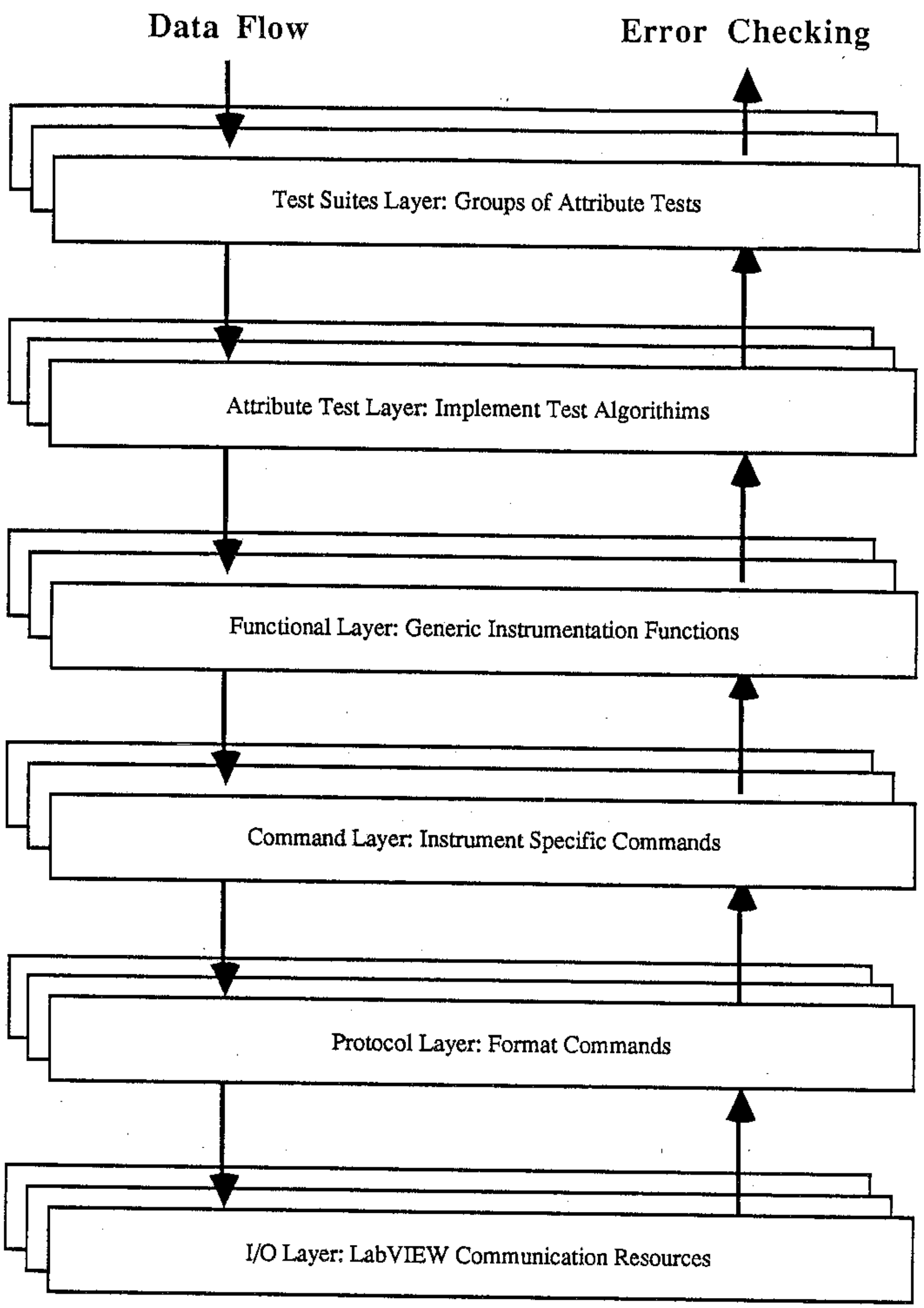
So far in our discussion of test network design and implementation, we have been addressing how to break your application into smaller, easy-to-manage pieces. Now though, we will turn our attention to putting the pieces together again.

The method we'll be using involves the creation of a software hierarchy. This is a methodology in which certain parts of the system "use" others in such a way that the system, as a whole, is simplified. Exactly how we do this is of prime importance since in a properly designed hierarchy, a fully functional, testable subset of the system is available to the user at every level. Needless to say, if you're working on a time-critical project (and who isn't), this can be a significant benefit.

The first thing we need is a model or framework, upon which we can build our higher level structures. The one used in the development of the automatic test station at Codex is shown in Figure 1. This model was developed over a period of about two weeks by Gary Mowry, another engineer at Codex, and myself. I had been using the basic concept of "levels" for months, but it was Gary who had the computer science experience to start nailing down what the levels were and how they were to be described. The chart before you is the result of our labors. Despite (or perhaps, because of) it's simplicity it has become a powerful tool for visualizing and evaluating ideas and potential implementation methodologies. In addition, it has served as an ideal model for presenting our work to higher level management.

To facilitate our discussion, I will approach the model from two different directions. In order to familiarize you with the intended functionality of each layer, we will start by examining the model on a layer-by-layer basis from the bottom up. Following this, I will present some suggestions on using it as an aid in system implementation in a top-down fashion.

But first, some general comments which highlight the model and the relationships between it's layers. (By the way, there is nothing magical about 6 layers. Depending on your application, you may have more or fewer layers.)



Common Automated Test Station
Software Hierarchy
Figure 1

One of the first things you will notice is that a well designed hierarchy blurs the distinction between applications and subroutines. For example, if you are a low-level driver looking up the hierarchy, all you see are more and more powerful layers of applications. Conversely, as a high-level test looking down, all you see below you is layer upon layer of tools and resources. Therefore, since all instruments at a given level are both constructed from lower level ones and potentially used by higher level ones, it is as correct to say that they are all applications as to say that they are all resources.

Another way this could be viewed is as a series of what I call "what/how pairs". At each layer boundary, the instrument above the interface specifies what function will be performed while the one below defines how it will be accomplished. This process continues down the hierarchy with each "how" being seen as a more specific "what" by the layer below it.

Finally, consider parameter passing. As a command flows "down" the hierarchy, the data used to express or implement that command increases in volume while narrowing in scope. The end result of this process is a large number of instrument transactions that are highly specific in nature. Likewise, responses being passed "up" the hierarchy undergo data reduction thereby becoming more abstract in make-up.

4.2 Layer Functional Descriptions

I/O Layer

This layer consists primarily, of the built-in LabVIEW Functional/Procedural resources. There does exist, however, the possibility of expanding this layer.

An example of this type of expansion can be seen in the way in which I communicate with multiple serial controlled devices. Since the Macintosh only has one serial port available for this type of application (the other one being tied up with AppleTalk), external hardware is needed. I choose a device which is basically a 9x1 RS232 multiplexer. Using this box, I am able to create up to 9 virtual serial ports on the Macintosh. While the resulting drivers are obviously complex instruments, functionally they are at the same level as the built-in serial drivers.

Other potential areas for extension might include:

- Support for AppleTalk (printing, file transfers).

- High-speed network interface cards and protocols (EtherNet, TCP/IP, NFS).
- Enhanced Toolbox access.
- Co-processor support

Protocol Layer

Instruments at this level implement device-specific communications protocols. Examples of this are functions that perform CRC calculations or poll for command errors.

These instruments are very useful for troubleshooting and system bring-up since they allow the operator to send commands directly to the desired device without having to worry about a lot of the communications over-head.

Command Layer

At this level, the function-specific command strings based on the device-specific syntax rules are built.

This is the level that most people are thinking of when they talk about writing "Instrument Drivers". There currently appears to be two schools of thought concerning how much should be included in these routines. One school feels that a single VI should implement all the functionality available in the hardware. The goal of this approach is to simply provide the system operator with a remote front panel for the instrument. The problem is that the front panels (and therefore the programmatic interface) for the resulting VIs have a tendency to become overly complex. This in turn, results in a larger number of parameters being passed, and hence, tighter coupling between components of the system.

The second school of thought, of which I am a member, asserts that the instrument itself should be subjected to the same type of functional decomposition as the rest of the system. While this results in a larger number of drivers, each one is simpler, more precisely defined and easier to maintain.

To see better how the two approaches contrast, consider the driver implementation for the TAS Model 1010 Line Simulator. This device is used in the telecommunications industry for simulating various types of telephone lines, as well as inserting common forms of signal degradation (i.e. Gain and Phase Hits, White Noise etc.). My first attempt was to put everything in one VI. The resulting front panel had a total of 25 controls distributed between 9 clusters. Needless to say this would be very difficult to use effectively. So at this point, I reexamined the instruments control structure and realized that the various functions of the instrument were operationally independent. Taking advantage of this, I developed a "family" of drivers to cover the capabilities of the instrument. Each of these drivers controlled only one thing and had, at most, no more than 3 to 4 controls on their front panel. In addition, since they were built on existing Protocol Layer routines, development time (including troubleshooting and debugging) was on the order of an hour or less per driver.

Functional Layer

Up until now interactions with the hardware have been very device-specific. You were dealing with a specific model of a specific type of instrument, manufactured by a specific company. Now however, you will begin taking high-level control of your test network. One of National Instruments foremost LabVIEW trainers, Dia Soubra, has been referred to this as the "do-er" level. I like that definition because you find at this level VIs with names like "Frequency Adjuster" or "Voltage Measurer". Since no mention is made as to how these functions are be accomplished, it can be seen that equipment definitions are beginning to get ambiguous.

There are basically two situations where this type of abstraction can be of benefit. First, let's say you had designed your program to use a Digital Voltmeter made by a certain manufacturer. Let's further suppose that one morning you came in to work and it had failed overnight. Normally, you would have to find another instrument of the same type to get your station working. If however, you had a VI with the same name and connector pane but designed to control a different meter you could simply install the different hardware in the rack, reload your program with the new VI and be back on-line immediately.

A second situation is found in my current test station. In this case I needed to be able to make a measurement that wasn't directly supported by any of the instruments to which I had access. I needed a way of measuring the time delay between the initiation of transmit carrier and a Carrier Detect signal on the modem becoming active. Likewise, I had to measure the corresponding function when carrier turned off. While the off-to-on transition wasn't a problem, to do the second measurement with a conventional counter required a way of triggering on the last cycle of a signal which is essentially a sine wave -- not a simple task.

My solution it was with a digital oscilloscope masquerading as an interval counter. Using the oscilloscope, I was able to trigger on the edge of the Carrier Detect signal and capture a trace showing the start or end of transmit carrier. It was then a simple matter to process the waveform to determine the time delay. As far as the rest of the software (or operator for that matter) was concerned, the required measurement was derived from a standard interval counter.

Attribute Layer

When we reach this layer in the hierarchy, we finally have all the tools we need to begin implementing test algorithms. If the software is structured correctly, this is also the first layer where the customer can begin getting some use out of the system. The goal is for each Attribute Test to be an autonomous entity, the function of which is to test one specific attribute of the subject. A pitfall to be aware of is the tendency to try to implement too much at this level. A good way to avoid this problem, is to write down the function of the test and ask the following two questions:

1. Does the description need to be a compound sentence?
2. Does it use words involving time, such as "first", "next" or "then"?

If the answer to either of these questions is "Yes", take a second look. There is a good chance that the test is actually a compound function that would be of greater value if further decomposed into it's separate pieces.

Test Suite Layer

A Test Suite consists of groups of attribute tests which, together, perform a complete functional test.

This level of instrumentation tends to be much more fluid than the previous ones because the end users of the station can (and generally does) modify it by adding or deleting attribute tests as they see fit. In fact it's not uncommon for end users, themselves, to begin developing custom Test Suites to satisfy new requirements you had never even imagined.

Summary

So now we are at the top of the pile -- or are we? Truly, it all depends upon how you have defined the purpose of your station. Perhaps you have (or will discover) a need for Meta-test Suites -- super programs that test an entire network of devices.

Maybe, as in the case of the manufacturing groups at Codex, your test station will itself be networked with a factory automation system. This results in a different kind of expansion in that the whole station effectively becomes a "test instrument" for testing a product or process.

In the introduction of Douglas Hofstadter's excellent book, "Gödel, Escher, Bach: An Eternal Golden Braid", he describes a canon written by Johann Sebastian Bach that manages to change key invisibly six times only to end up back in the key in which it started (but one octave higher). This is analogous to the software hierarchy I have just outlined -- as you work your way up through the levels to ever higher and more remote "keys" you can suddenly find your self back where you started, with an instrument and a requirement to be fulfilled. The lesson is simple -- never assume that you have reached the top of the hierarchy. There is always more that can be done.

4.3 Model Utilization

Recently a fellow engineer at Codex walked into my cubicle and told me that he had been assigned to evaluate the test options for a new project. After a short conversation, it became clear that the Test Station, as it currently existed, would provide about 85% test coverage for his product while an additional 10% was do-able with a few minor modifications to the station. By the end of the conversation, the only question he had left was: "Where do I start?". With this, I hand him a copy of the Hierarchical Model and said, "At the top...'.

Test Suite Definition

The first thing to do is make a list of all the attributes of the test subject that will need evaluation. I have found it useful to get as many people as possible involved with this process. Depending upon the size of the project you might even want to have a few formal brain-storming sessions. Of course it goes without saying, that any applicable standards or specifications will be canvassed for testable attributes as well.

A key point to remember in developing this list is that as you start work on a new project many attributes can be "To Be Determined" or "TBD". Be careful to isolate these functions as much as possible from the remainder of the system, so that as they evolve the impact on your schedule is kept to a minimum.

Once this list is complete, start grouping together properties of the subject that have to do with the same general area of functionality, like all the loopback tests or all the transmitter characteristics. As this process continues you will begin to see four or five of these groups develop. Repeat this process several times, experimenting with different groupings. When you are finally satisfied with the combinations, these will be your Test Suites.

Attribute Test Definition

Next, taking each group in turn, analyze them to determine which subject attributes go together more directly, like transmit level adjustment and transmitter output spectrum. As before, repeat this process as many times as necessary to end up with a logical set of groupings. These will be your Attribute Tests.

Functional Layer Definition

At this point, you may again want to bring in some outside help, because you now have to define the methods you are going to use to perform each test. This description has two parts (1) a description of the algorithm to be used, (2) a list of resources (functions) needed to implement the algorithm.

If you are working on a mature station (one that is already being used for testing something else), be conscious, as you develop this list, that some of these resources may already exist. If you aren't blessed with the availability of a mature station, remember that you may need similar functions in other places, so think generically. Think reusability.

Command Layer Definition

With this list of resources in hand, determine the best means of implementing these functions with the hardware available in the station. Among the things that should be consider are required accuracy, data acquisition speed and instrument availability.

Protocol / I/O Layer Definition

Finally, with the function defined and the appropriate instrument selected, you are ready to begin getting the commands out to the instruments. An important way of speeding the development time for a station is to, based on past experience, identify the probable complement of test instruments as early as possible and have someone working on the lower level drivers for these units in parallel with the rest of the design effort.

At this point, the ground work is finally completed and you are ready to begin working your way back up the hierarchy in code. If you have done the work correctly, you should be able to hand the design documents that have been generated (you have been writing this down haven't you??) over to a team of implementers for actual code development.

5. Conclusion

So in summary, we have covered three of the major factors influencing the development of large LabVIEW test networks. Before I finish though, I wish to one brief philosophical side trip to examine some of the personal effects of this development project.

The first thing I have noticed is a change in the way I think. I have come to realize that the concept of Functional Decomposition based on Information Hiding is an excellent paradigm for problem solving in general. Used properly, it can produce a clarity of communication that will have a revitalizing effect on your interactions with others.

Secondly, my understanding of the importance of hierarchies has deepened greatly. When I started, I saw that visualizing the test station in terms of a layered architecture allowed it to be understood more easily. This in turn showed me how the station might fit into a larger hierarchy of factory automation, but beyond that I was still thought it was strictly a tool for Computer Science. That was about the time I stumbled across the book, "Gödel, Escher, Bach: An Eternal Golden Braid". This book started me thinking about hierarchies in general and I gradually came to see that the reason that a hierarchical system seems to work so well for visualizing large systems is that it was apparently the design methodology used in the development of the largest possible system -- the world in which we live. If you think about it, we are all imbedded in a multitude of interlocking hierarchies based on ecology, family, business and religion. So while I don't presume to know why hierarchies are so common, it seems to me that for any pattern to be in such general use it must touch on some nerve of that which is true and universal.

6. Acknowledgements

I have been greatly aided in the preparation of this paper by the entire Common Automated Test Team, in particular Gary Mowry has provided invaluable help in the clarification of the issues presented here. In addition, I am extremely thankful to both Mike Reed and Norm Dube for the parts they played in critiquing the paper prior to publication.

Appendix A

1. Dijkstra, Edsgar, W - "The Structure of "THE"-Multiprograming System", Communications of the ACM, May 1968

2. Parnas, David L. - "A Technique for Software Module Specification with Examples ", Communications of the ACM, May 1972

3. Parnas, David L. - "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, December 1972

4. Parnas, David L. - "Information Distribution Aspects of Design Methodology", Proceedings of IFIP Congress 71 - Volume 1

5. Parnas, David L., Darringer, John A. - "SODAS and a Methodology for System Design", Proceedings of IFIP Fall Joint Computer Conference, 1967

6. Stevens, W.P., Myers, G.J., Constantine, L.L. - "Structured Design", IBM Systems Journal, Vol. 13, No 2, 1974

7. Parnas, David L. - "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, March 1979

8. Berzins, V., Gray, M., Naumann, D. - "Abstraction-Based Software Development", Communications of the ACM, May 1986

9. Zemanek, Heinz - "Abstract Architecture - General Concepts for Systems Design", Winterschool of Abstract Software Specification - Copenhagen, February 1979

10. Brodie, Leo - "Thinking Forth - A Language and Philosophy for Solving Problems", Prentice-Hall, 1984, Paperback

11. Hofstadter, Douglas R. - "Gödel, Escher, Bach: An Eternal Golden Braid", New York: Vintage Books, 1980, Paperback