Zero to Hero: Designing the LabVIEW Drivers for Your Instrument

Jim Cavera, AIM-USA, LLC

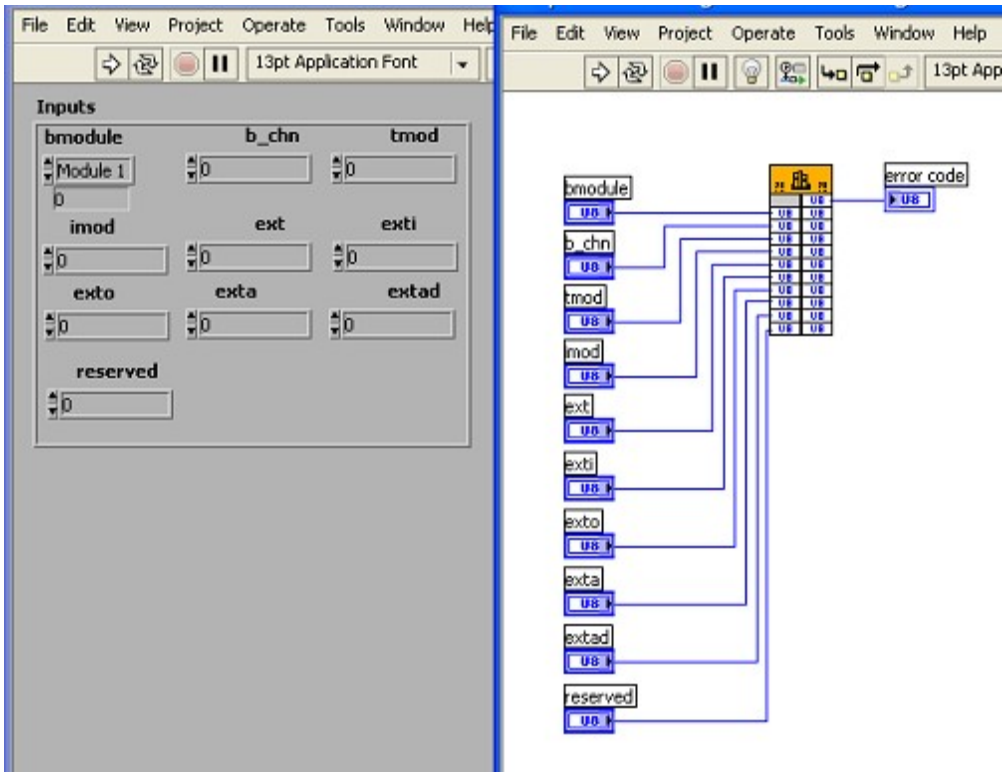
Introduction

Let us suppose that you work for a company that is developing a specialized piece of data acquisition hardware. You've been given the unenviable task of developing the LabVIEW drivers for the brand new model. All you have to go on is a DLL and a set of C headers.

Alternately, your company has an established niche and set of instruments, but little-to-no LabVIEW support. Someone coded some, at one point in time, probably in LabVIEW 7. You know that the lack of up-to-date drivers is hurting your position and you feel that you must do something about it.

What a bad driver looks like

Bad drivers take many different forms. The obvious first tactic that most companies take, is to map all of the functions of their DLL directly to LabVIEW library nodes and call it a day. There's often no attention paid to data types, error detection, comments, or even clean wire placement. Documentation is non-existent with the developer believing that the API manual (full of C goodness) is perfectly reasonable. And so most first attempts at drivers, end up looking like this:



Care to guess as to what the parameter names mean? Or the error code? Granted, this is a fairly common first-draft, but it is not to be mistaken for a proper driver (or even a proper VI call). And of course, there is one of these for each and every public function in the DLL. And nothing more. There is no documentation associated with the VI, no help, no explanation of the inputs and outputs, and no

error detection. An end-user would have to be very familiar with the underlying code in order to build a system with these components. And in the LabVIEW world, this is unacceptable.

Tackling the problem

But for all of the bad, this is not an impossible task. As with all engineering work, before we begin, we have to define what we want. And I'll start by laying some ground rules.

> Rule 1: All LabVIEW code should be written for three levels of user.

Of any system, there are three types of user. The first is the beginner. They've no real idea (perhaps just a vague notion) of what they want the system to look like. For them, we must provide examples that they can look at. These examples will be ready-made solutions to common use-cases for the instrument and could be used out-of-the-box, or with very little modification.

Then there are the intermediate users. These individuals have a pretty good idea of what they want to do, but they want to take very little time to do it. The examples will be too simplistic for them and so they'll need high-level functions that they can assemble in meaningful ways.

And finally, there are the power-users. They will welcome the ability to push every bit in and out of the DLL. They will be making use of the low-level functions, but should not be constrained by them. For these users, we must provide all of the functionality of the underlying DLL, but with all of the LabVIEW niceties. Though more advanced than the others, they will *not* want to be cleaning up our messes for us.

These various stages of development will take place in several phases, and in more-or-less the reverse of the aforementioned order. The advanced users will get VI's that are all but DLL wrappers. And then we will code the intermediate layers (yes, there will be more than one intermediate layer). Finally, the examples will be developed for the beginning user.

Phases of development

Phase zero will consist of preparing the DLL. This is *not* phase one, as LabVIEW will not be entering in to the equation, except tangentially. DLL's are messy and C calling conventions do not necessarily match what LabVIEW likes to see. We will begin by correcting this. And please note that, if you can avoid the messiness of a DLL, feel free to skip step zero. The LabVIEW VISA tools are extremely powerful and can drive most types of instrument without resorting to low-level calls.

Phase one is when we begin the LabVIEW coding. And yes, we shall be doing a one-to-one mapping from the DLL to the VI structure. Additionally, we begin to add things like error detection, type and range checking, and code documentation.

At phase two of our development process, we begin to abstract from the DLL. This is where we write code that aggregates common tasks. Additionally, we provide programming tools for things like data conversion and error logging.

Phase three builds on top of two and is a continuation thereof. This is where we write templates, polymorphic VI's, and objects. In many respects, this is the most critical phase of the design, because it is what the beginning and intermediate users will see when they look at the sample code.

Finally, phase four is the sample code and documentation. And, like good programmers, we always do documentation as we go (right?).

Phase zero in depth

This is going to be a big time sink in terms of the development effort, but it is a necessary step. No doubt, your DLL has functions that look like this:

```
AiReturn _API_DLL_FUNC AiFcCmdMonTrgWordIni (AiUInt32 ul_Handle,
                    const TY_FC_MON_TRG_WORD_INI *px_MonTrgWordIni );
```

And that is all well and good for those who code in C. But what does this mean to LabVIEW? We have a handle that may or may not point to a block of memory; and we have a pointer to a structure that itself could contain other pointers or other structures. This is not a desirable condition in the LabVIEW world. And in fact, you would probably bring your development environment to a screeching halt were you to call this function directly. To rectify this situation, I am going to recommend three things, each of which will annoy both your manager and the DLL development group.

1. Write your own DLL.
2. Use only freely-available development tools
3. Give away your source code

For the first, in writing your own DLL, you can provide a LabVIEW-friendly wrapper to the C-centric functions of the "official" DLL. Since you control it, you can put everything in to native data types and save yourself a lot of work when it comes to coding the low-level VI's.

For the second and third, by using freely-available tools (Visual Studio Express, Eclipse, etc.) and giving the code away, you turn your power-users in to developers. Note that, since you are only giving away the wrapper code, you are not divulging any company secrets. And the power-users of the world become your allies in the quest for the perfect interface.

So now that you've managed to upset your manager, it's time to lay down some serious ground rules for DLL development. There are a few good guidelines scattered about the NI Knowledge Base, but in general you should strive for a few of the more basic ones.

First, only pass data that LabVIEW understands. In the following example, I've listed a "before" and then a part of the "after" to illustrate this:

Before:

```
int AiFcInit (char** serverlist);
```

After:

```
int32_t LV_AiFcInit (LVStrArrayHdl *pServerList);
```

Is it possible to pass data as a "char**" type from within LabVIEW? Of course. But if you wrap it as an "LVStrArrayHdl" its use becomes much more obvious. Likewise, there is no real problem with using an "int" as a return type, but by using one of the LabVIEW-defined types, the size of the variable is obvious. And again, you are not divulging company secrets. The code in your wrapper DLL would look something like this:

```
_API_EXTERN int32_t _API_DLL_FUNC LV_AiFcInit (LVStrArrayHdl *pServerList)
{
	int32_t		result = 0;
	TY_SERVER_LIST *slist = (TY_SERVER_LIST *) DSNewPClr(max_boards * sizeof(TY_SERVER_LIST));

	result = AiFcInit(&slist);
	if (result == 0) {
		int i = 0;
		do {
			int j = 0;
			while ((slist->auc_ServerName[j]) && (j<MAX_SERVER_NAME_LEN)) {
				(*((*(*pServerList))->elt[i]))->str[j] = slist->auc_ServerName[j];
				j++;
			}
			(*((*(*pServerList))->elt[i]))->cnt = MAX_SERVER_NAME_LEN;
			i++;
			slist = slist->px_Next;
		} while ((slist->px_Next) && (i <= max_boards));
		(*(*pServerList))->dimSize = i;
	}
	DSDisposePtr((void *) slist);
	return (result);
}
```

It does, of course, take a bit of work to translate the data types. You need to do things like make deep copies of strings, pre-allocate and then properly dispose of memory, and make sure that you return meaningful result codes. And this brings me to the second rule:

> Rule 2: Decide on a memory management strategy and stick to it.

As I'm sure you've found out at some point, most of the errors you will encounter in C, are memory management errors. The interactions with the LabVIEW environment only serve to compound these issues. And so it is best to decide on a strategy early and stick with it. Some example rules:

- All memory must be pre-allocated by the calling VI.
- All arrays used internally to the DLL are created and destroyed within the wrapper.
- All strings must be pre-initialized to null strings of the maximum possible size.
- All relocatable memory must be locked while in use.
- Always make deep-copies of strings prior to passing them.

These are just a few examples. I'm sure that you can come up with many more.

And while we are on the subject of memory, there is a special case that I would encourage you to explore: static state memory. It may, during the course of your development, become handy to track the state of a system and maintain that state data from within the DLL. This is fully possible and not nearly as scary as it may seem.

In my own wrapper DLL, I have declared the following structure:

```
typedef struct _statedata {
	char	files[16][256];		// image file names
	int	imageIndex;		// index to the current image
	int	numImages;		// total number of images
	int	version;		// wrapper DLL version number
} StateData, *StateDataPtr;

static StateData mystate = {{"","","","","","","","","","","","","","","",""}, 0, 0, 0x1002};
```

Note that this has a few fields that I want to track: names, index, and number of names. And then there is a fourth field: version number. If you intend to store state data within the DLL, I would urge you to add that field. This is the version number of the wrapper DLL itself, and you can provide an accessor function for it from within the LabVIEW code. It makes things easier if you wish to continually provide updates to the main DLL and the wrapper DLL, both.

And of course, if you do provide for static data, you'll need an initialization function and the accessors for each member of the static structure.

Finally, a few brief notes on style. I would urge you to give your wrapper functions, names similar to those in the underlying DLL. In my case, I just used the underlying function name, with "LV_" tacked on to the front. This keeps it easy for those pesky power-users.

And remember, follow the golden rule of coding: Document for others, as you would want them to document for you. I don't know of anyone who actually enjoys reading someone else's code. At least try to make an already painful process, less painful.

Phase one – starting the LabVIEW coding

Assuming that you now have a nicely-organized DLL (with LabVIEW-native data types), it's time to wrap those functions in to VI's. Again, phase one is where we do the straight, one-to-one mapping. Every function in the wrapper DLL (barring things like internally used functions and call-backs) will get its own, corresponding LabVIEW VI.

This phase is just as critical as phase zero because of three factors: we do type and range checking, we do error detection, and we provide meaningful documentation by way of user-intelligible variable names and comments.

But prior to the niceties, we have some serious business to take care of. Just because the wrapper DLL functions are in a LabVIEW-friendly format, it does not mean that you can still call them safely. Two things to watch for: calling conventions and memory allocation (no, we're not yet free of potential memory bugs).

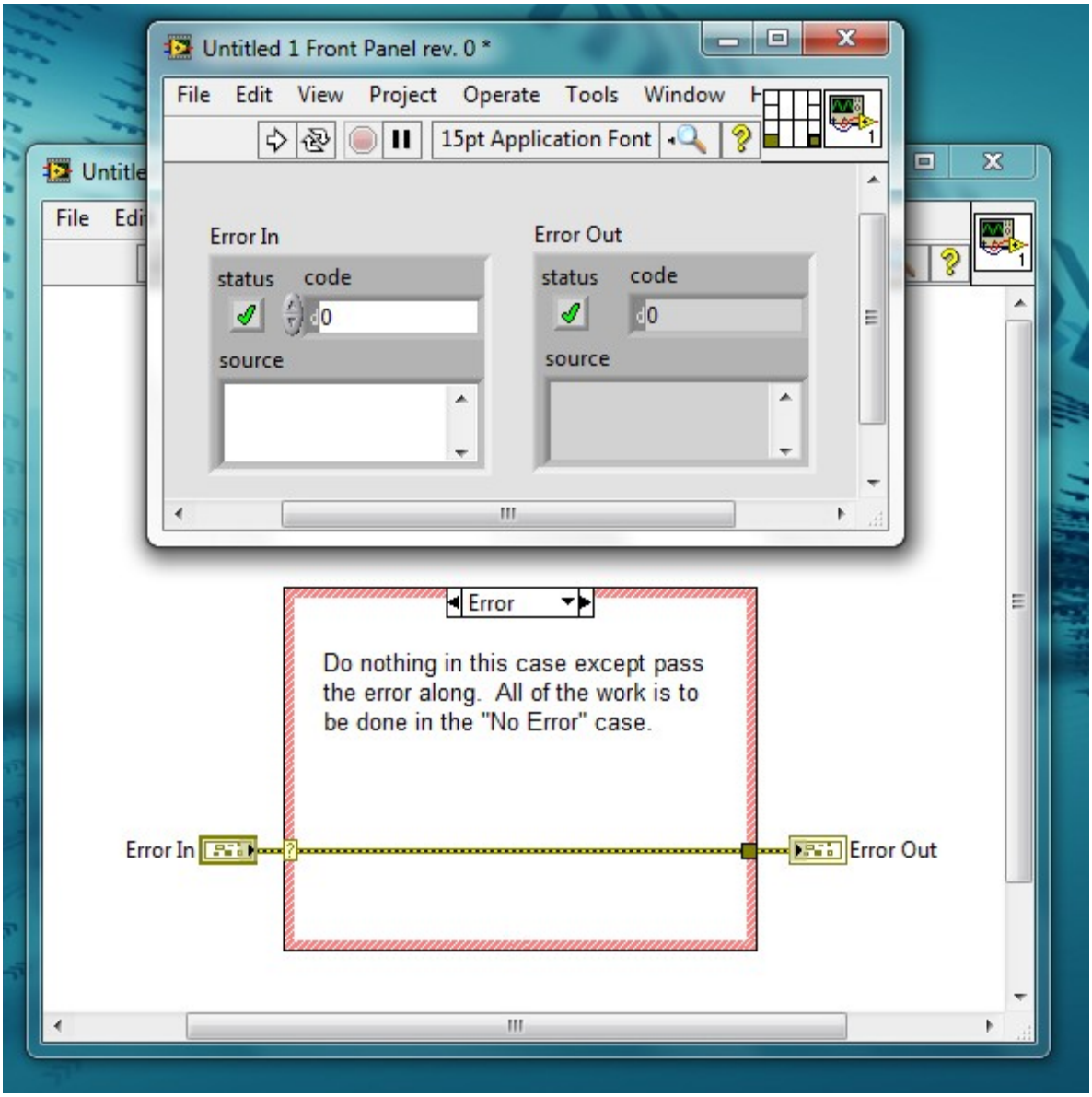
Calling conventions first. Note that in the "call library function" node (henceforth, simply referred to as a "library node"), there is a choice of calling convention: standard C, or Windows API. Choose wisely! If you are writing your wrapper DLL in Visual Studio version whatever, your code will default to the Windows API convention. If you are using Eclipse; standard C. And of course, you can set the calling convention yourself in the linker settings. Whichever you choose, pay attention to it. Choosing poorly will cause your system to crash, lock up, or blue screen.

And then there's the issue of memory management. Remember the rules that you decided upon? It's time to double-check yourself and make sure that you are following them. If in coding the wrapper DLL, you determined that all memory needs to be pre-allocated prior to entry to the library node; you'll get a nasty shock if you pass an empty array and then try to fill it. These bugs can be both subtle and maddening.

So, bearing those factors in mind, doing the initial wiring of the library node should be simple. After all, you've already done much of the work in the wrapper DLL. Now it's time to add in the error handling.

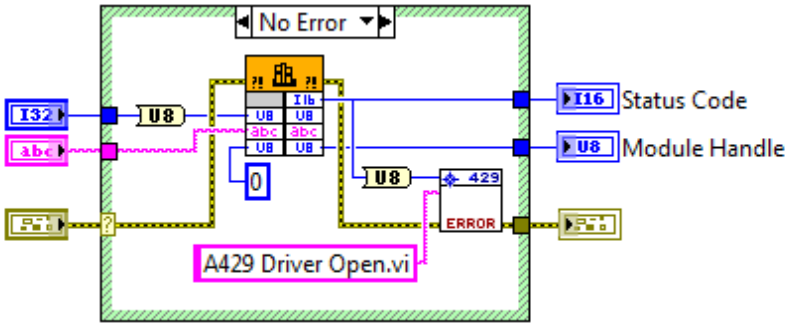
As you are probably aware, LabVIEW provides a lot of built in error processing capabilities. The obvious is the condition of a fatal exception: you have the pleasure of seeing a debug dialog just prior to LabVIEW failing completely. In our error handling, we try to eliminate this possibility as well as providing meaningful feedback to the user.

The first priority should be handling an existing error. In this case, an error has been generated by another VI in the chain. All we need to do is pass it along and (probably) not try and perform any action ourselves.



Note a few things about the above VI: first, the “Error In” and “Error Out” controls are placed in a standard configuration with respect to the VI terminals. You should strive to always use the same terminals for every VI that you write. This makes things easy to connect together and makes the finished diagrams look a lot neater.

Second, the case structure in the wiring diagram can directly accept the error data type, creating cases for “Error” and “No Error”. Use the error case to pass through the error information without doing any processing. All VI actions will take place in the no-error case. Again (just to reiterate), do these for every VI that you write.



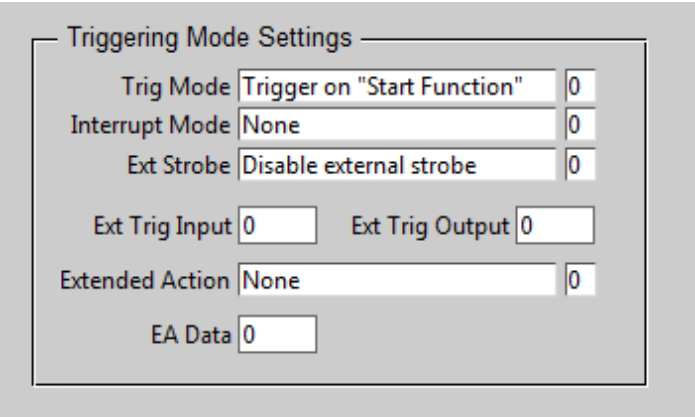
Note the structure of the VI shown above. It has all action taking place in the “No Error” case. The only thing that happens in the “Error” case is to pass the existing error code and set the VI outputs (status code and module handle) to some harmless values. Also note that this consists of a single API function call. For the phase one VI's, we do the one-to-one mapping of the DLL functions to the VI's.

Notice also that we have another sub-VI call. This is part two of our error handling strategy: providing meaningful feedback to the end user. In our error handling VI, we take as inputs the name of the VI and the result code from the function call. We use those to populate the error cluster to produce a message that may look something like this:

> Warning. Error in “A429 Driver Open.vi”: Unknown server name.

This saves the programmer from having to look up a potentially-meaningless result code. If you wish, you can do fancier things with the error handling sub-VI. I would advise defining a global flag to log errors to a text file. That way, if an end-user does experience a fatal crash at some other point in the call chain, they can see the last error that was generated by your VI suite.

Though not explicitly shown in the above example, phase one development is also the appropriate place to do things like data conversion and range checking. Ideally, you want to make it impossible for an end-user to feed invalid data to the DLL function call. You can do this by performing range checking on numeric data (and generating an appropriate error for the out-of-range condition) and making good use of enumerated constants for values that must lie in a particular range. Here's an example of what such a control cluster would look like:



Notice that some of the inputs to the library node are represented by enumerated constants. These can be converted to integers as needed by the library node. They serve to limit the range of the inputs and to provide meaningful labels for the end user. In this way, they user doesn't have to know that a value of two means “enable external strobe”.

As for range checking in this example, you could make sure that the trigger input and output channel numbers are set to an acceptable value (0-7, for instance) and return an error code if not, all without calling the library function and potentially incurring a fatal exception.

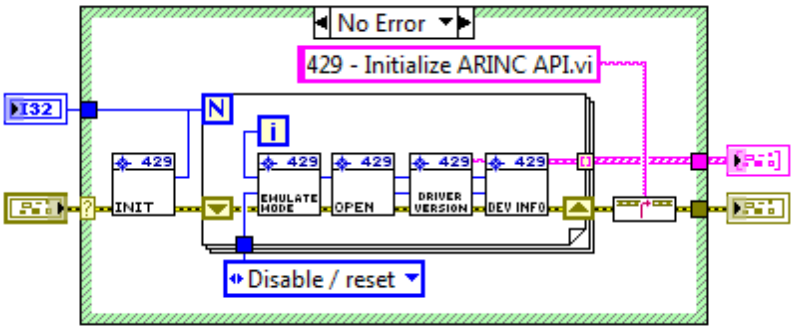
On to phase two – function aggregation

Phases zero and one of this process consisted of reusing and intelligently organizing what you already had: namely, the C level API function calls. In phase two, things start to get much more interesting. The goal of this phase is to provide some higher-level support for the user by aggregating common function calls. If, for instance, you find that you must always call four functions in a particular sequence before making use of the instrument, those four VI's should be aggregated in to a single initialization VI. This is what phase two is all about.

Of course, this is highly dependent on the precise nature of your instrument. Let us take a simple case and say that you are designing a bus analyzer. In order to properly encapsulate common activities, you must first identify what these common activities actually are. In the case of a bus analyzer, they might be things like this:

- Initialization of the instrument
- Initialization of the bus monitor
- Setting up a trigger condition
- Starting and stopping the monitor
- Reading the acquired data
- Gracefully shutting down the instrument

All of these deserve to have just a single VI that is obvious in design and structure to the end-user of the system. Let's begin with the first action as an example.



In this VI, we have several phase one VI's working in combination. The first is the basic initialization call. This returns the number of devices found in the system (if more than one). Next we enter a loop, calling several VI's once per device. The specifics of those VI's in this example is unimportant.

But notice a few things: first, we are still following our error-handling convention. Each sub-VI contains its own error handlers, but the system as a whole also manages the error information. As a corollary to this, note that the error handling sub-VI itself, looks different. In the previous example, we saw a VI that took an error code and a VI name as inputs. In this example, we've done away with the error code, since each sub-VI provides its own. Instead, we take the VI name and tack it on to the existing error message in a meaningful way. The same error message as before, could be modified to look like this:

> Warning. Error in "A429 Driver Open.vi": Unknown server name.
> Sub VI of "429 – Initialize ARINC API.vi"

In that way, the user knows right where to go when the error message pops up. Again, once you have decided on an error-handling strategy, it is important to stick with it.

As far as our error handling goes, you may have noticed that we let the loop run in the event of an error. This can easily be remedied via the use of a stop terminal in the loop, but there's really no need. Since every VI checks for errors (and does nothing if there is an error in the chain), the loop may continue to run, but would be completely ineffective, doing nothing anyway. In either case, the first error flag is the one that gets passed out of the VI.

Also note one other thing: All of our VI's are nicely aligned with wires running to them in a neat and logical fashion. Since we standardized on a set of terminals in phase one, we can make the program flow much more obvious to the user. And too, all of the VI's have a similar appearance: tiny logo at the top left, suite name (429 in this case) at the top right, and the function name in the lower portion. When developing a suite of VI's, it's the little things like this that make a huge difference in terms of usability.

Please bear in mind that there are some things that should *not* be aggregated. Keep these things separate under logical lines. For instance, if an instrument has multiple parts that can be used independently, it makes sense to keep the setup of each part completely separate. As a case in point, setting a real-time clock should not be a part of the "Init" VI, simply because the end user might want the instrument to run in GMT rather than EST. Always keep in mind what the user may and may not want to do. You're going to be coding a bunch of examples at the end to show off finished systems.

Save the major integration for the examples.

Now that you've gotten the idea of aggregation, we come to the second thing that you should be working on in this phase: documentation. Actually, you should have been working on documentation all along, at least in a minimal sense. Here's what you have to do in order to play nicely in the LabVIEW sandbox:

- Make sure all VI's have a meaningful name.
- Use the "documentation" section of the VI information to include at least a help tag and a meaningful comment. These show up in the on-line help for the VI and should be as complete as practical.
- Write a description and tip for each control and indicator on the front panel.
- Use the revision history feature to track changes made to the VI.

If you do these well, then you've accomplished a lot more than most VI authors. And you'll make the end users very happy. But there is one more thing that you need to be doing at this point.

A manual is an absolute must. Yes, your users will have the API manual for the DLL, but that is no replacement for a LabVIEW-centric set of documentation. The API manual is a good starting point, but make sure that your VI suite has its own users' guide, with VI connections listed, what they mean, what is require and what is optional, what any defaults are, and what any error codes mean. It is no understatement that your manual can make or break your product.

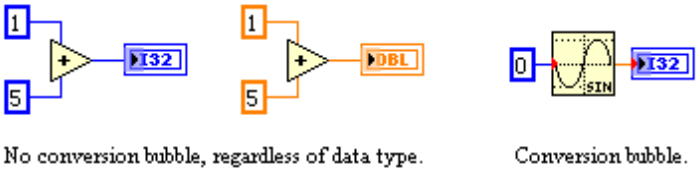
And finally in phase two, you need to provide programming tools. These are LabVIEW VI's that do *not* make use of the underlying DLL. If your instrument can write to a non-standard data file, then provide a set of tools to read, write, and convert that file. If you wish to include some sort of configuration management (remember the static data?), then provide a user interface such that those configuration values cannot be set to something invalid.

Though this may seem to be a small part of the driver suite, it is a very important one. Again, you will be judged on the quality of the tools that you provide. Even (no, especially) those that do not directly access your instrument.

Phase three – thinking like an end user

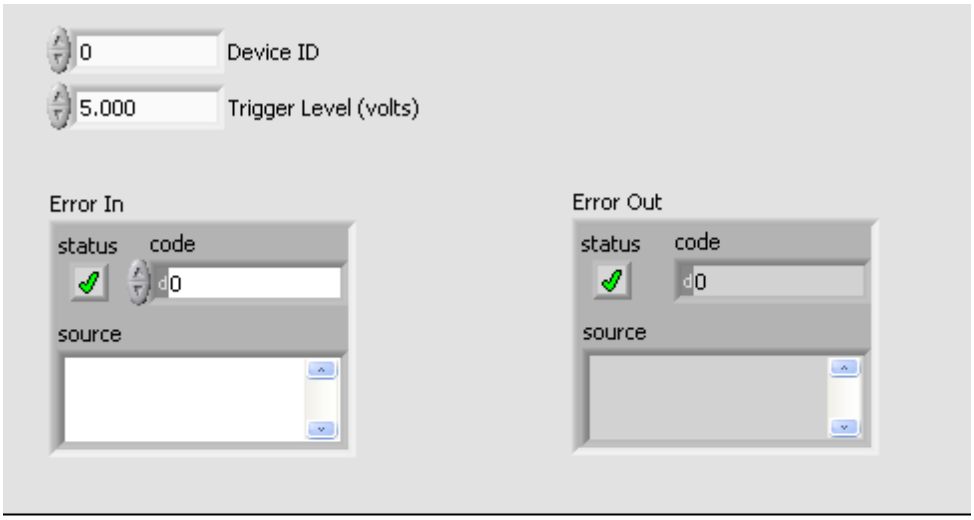
This is where we get down to business. In this phase, you must think like and end user and try to code what they want to see. This is where you get to play with things like objects, custom data types, and polymorphic VI's.

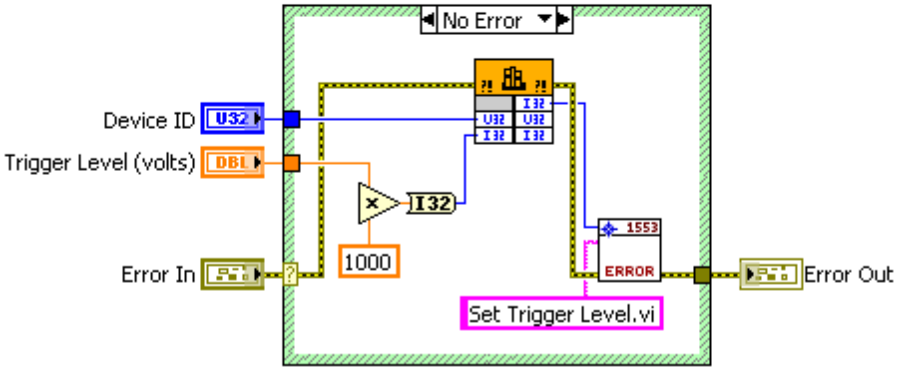
Polymorphic VI's are designed to take multiple data types as input; or spit out different data types depending on what is wired to them. You are probably familiar with this concept: the built-in VI for addition, can take any numeric data type as an input without showing a "conversion bubble". Other VI's require data to be in a specific type on input and output.



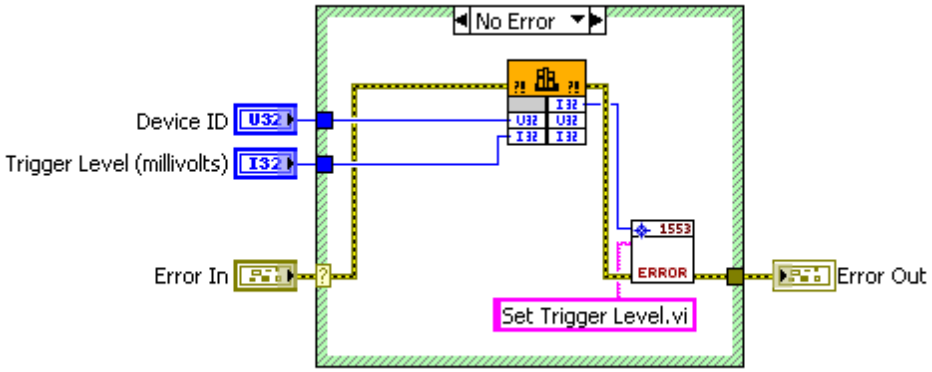
No conversion bubble, regardless of data type. Conversion bubble.

In your own VI suite, it can be desirable to have the ability of accepting data of more than one type for a particular input. Let's assume for a moment that you have a VI that sets a trigger level in volts. Your DLL requires that data in millivolts as an integer, but you would like to give the user the option to specify voltage as a float as well. You can do this by creating a single, polymorphic VI.

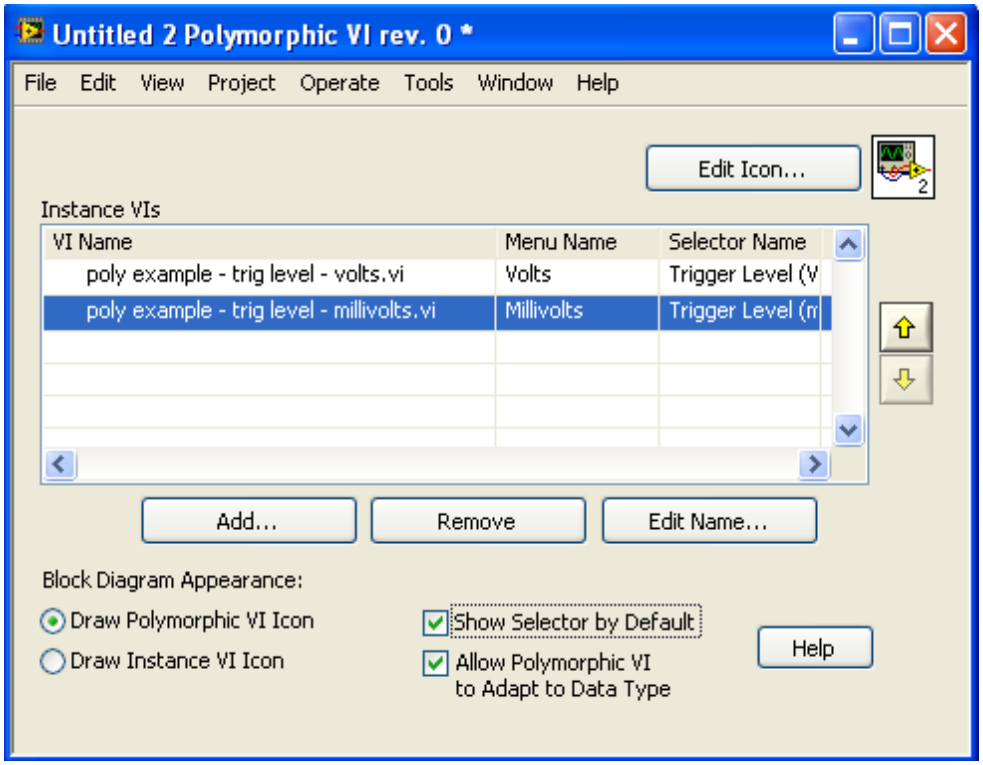




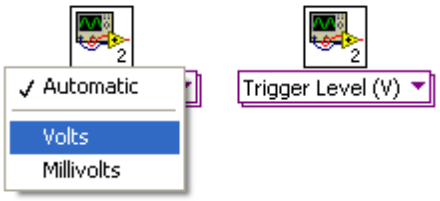
The creation of a polymorphic VI is done in two parts. First, you create the VI for each data type that you want to handle. Above, you can see the appropriate VI for accepting the trigger level in volts. The diagram of the VI to handle millivolt input is shown below:



Save these as separate VI's and give them two different, but meaningful names. Something like “Set Trigger Level – volts.vi” and “Set Trigger Level – millivolts.vi” would be a good choice. Once you have VI's for each of the input types that you want to handle, you can combine them in to a single, polymorphic VI. Just open an new instance of a polymorphic VI and add each instance VI to the list. Be sure to give each a meaningful name such that they can be distinguished in the menu.

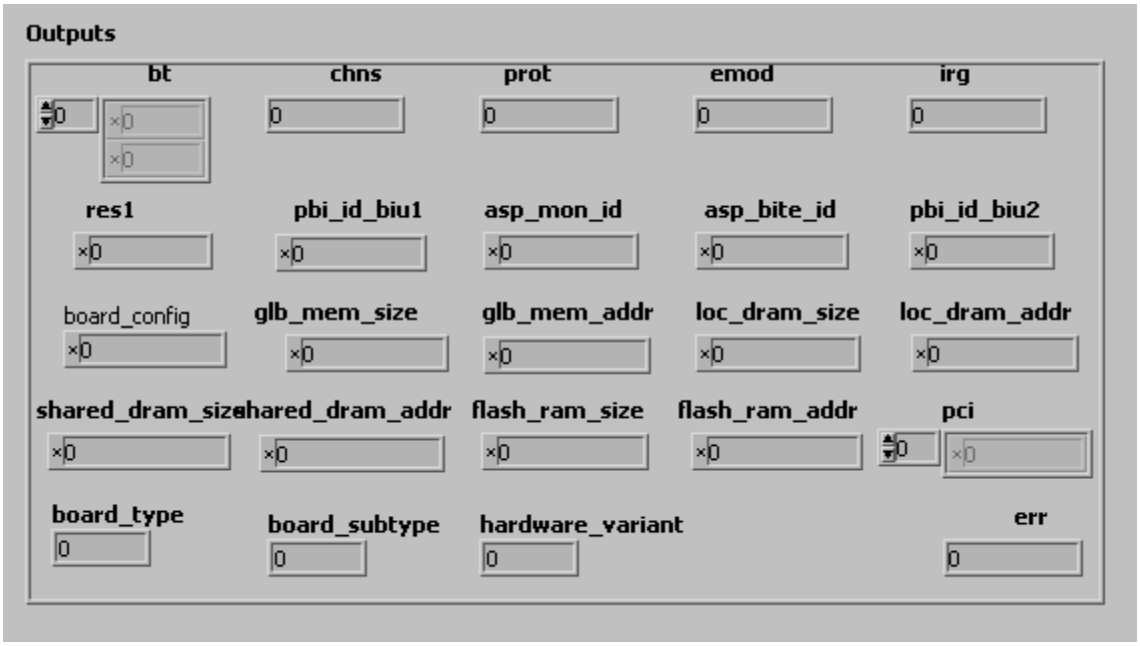


The user will see a single VI called “Set Trigger Level.vi” that magically accepts data in a few different formats and can be manually toggled between them.

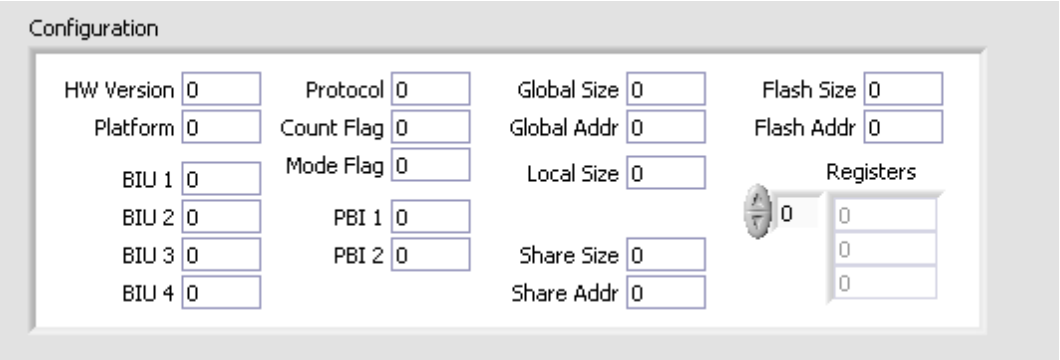


But polymorphism is just the tip of the object-oriented iceberg. The next step (and one that isn't object-oriented *per se*) is to provide for control typedefs. Though this is not at all required, it does make for a nice looking and easy to use set of VI's.

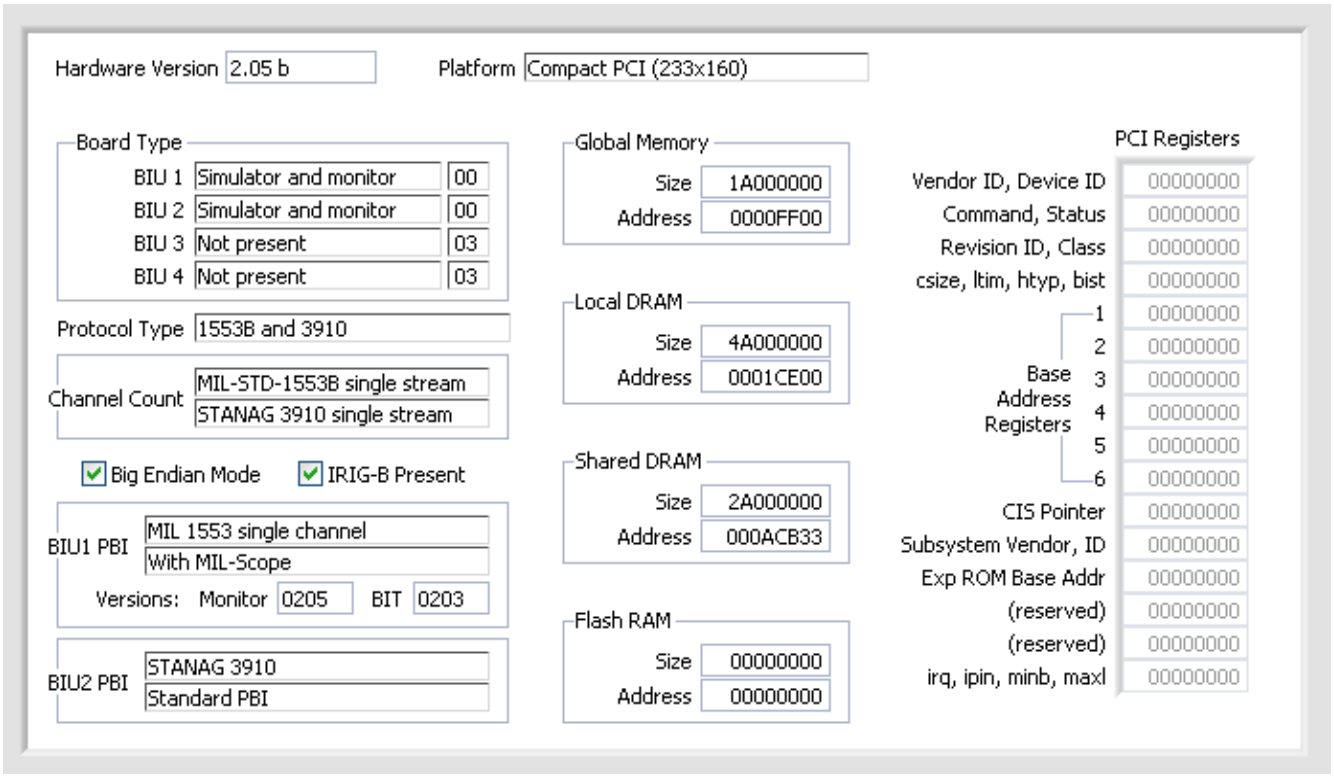
Let us suppose that you have a VI that returns a lot of configuration information from your instrument. There are dozens of parameters and each has dozens of possible states. You can (and perhaps the previous programmer, did) take the easy way out and put them all in to a cluster. But hopefully, you are a little bit better than that. Though this may have been the before:



Your “after” looks like this:



Which is not a bad thing, but it could be a lot better. Rather than displaying (or controlling) meaningless data, you should make use of the programming tools developed in the previous phase. Flags used for things like “platform type” and “protocol” could be translated in to meaningful text, rather than kept as numeric data. Take a bit of time to write VI's to convert as appropriate, with the end result being something like this:



Here, it's easy to see the meaning of all of those numeric flags. Displaying data in a cluster like this gives much better feedback to the programmer. And though it may seem like a perfectly good way to kill off a few hours of your time, your end users will thank you in the long run.

When designing displays and controls, it helps to have a few rules worked out ahead of time. Here are a number of good ones:

- All enumerated information is to be presented as text
- All data for which there are only two possibilities, is to be presented as boolean
- Data is to be categorized in to logical sections

You get the idea. If you need additional information on display design, I would recommend reading anything by Edward Tufte, and the original (1985) edition of the Macintosh Human Interface Guidelines. Both are considered classics in the field of displays.

So now that you have a beautiful set of clusters to display and control complex data, you will want to save those as controls. I strongly recommend saving them as typedefs rather than stock controls. If you are not familiar with typedefs, these allow a control to "subscribe" to the type such that, if you make changes, it propagates through to all VI's that make use of the associated control or indicator. Again, this is not necessarily an object-oriented piece of the puzzle, but it is very important to both the usability and the maintainability of the finished VI suite.

And now on to actual objects. I'm not going to go terribly deep in to this topic, as it is the subject for a full-length course. I will, however, get you started with some basics. In LabVIEW, objects look a whole lot like a VI suite in and of themselves, but with controls associated with that suite. The suite of VI's can be likened to the methods of the object and the controls, to the data.

Every time you instantiate an object, LabVIEW makes a new, internal, copy of the data; each copy being associated with that object. From the user's perspective, they never even have to see the data. The end user will (or perhaps, should) only see the methods associated. This solves a few problems for us. First, it lets us hide a lot of the inner workings from the end user. If you are creating a driver for an arbitrary waveform generator, you can define as one of your objects, a "waveform". From that point onward, the user can manipulate the waveform without having to know all of the potential ugliness of the underlying data. This is best illustrated via a simple example.

```
Waveform object
        Private data:
                uint32_t      data[size];   // array of all of points making up the waveform
                uint32_t      timebase;     // time in microseconds between individual points
        Public methods:
                Add Sine Wave
                Add Exponential Decay
                Add Sawtooth Wave
                Add Square Wave
                Offset and Scale
                …
```

As you can see, the end user can employ the public methods to build up a waveform, all without having to manipulate points, thus hiding much of the underlying complexity of the device. I would urge you to do a simple exercise in which you try to break down the functionality of your device in to logical objects, based on the expected needs of your users. In my own "objectification" of the avionics bus, I was able to identify unique objects for things like: data packet, data parameter, channel, etc.

It is *very* important to not go overboard with this. Object-oriented design is not a panacea, nor is it meant to be applied universally. Good VI design will include procedural code for everything and objects only where they make sense. Here are good places to make use of objects:

- A root object for the device itself (particularly helpful if a customer may use more than one)
- An object representing data output from the device (as in the arbitrary waveform example)
- An object representing the type of data acquired by the device (if complex)

And here are some things that you should *not* have as objects:

- An acquired waveform (LabVIEW already has a waveform data type – use that)
- A single, scalar measurement (an object for a number – no need to bother)
- A data file (again, use the LabVIEW types if at all possible)

You can probably sense a theme to this one: do not use objects in cases where National Instruments has already done the work for you. A "voltage object" makes no sense – it's just a number. File objects are already covered by the various LabVIEW file utilities. Even things like numbers with specific units attached can be handled adequately in LabVIEW. A good rule of thumb is, if you find yourself duplicating existing VI's in terms of functionality (but with an object), you may be doing something wrong. The goal of this is to make is simple for the end-users, not more complicated.

The final phase

Phase four is where you really put on the polish. This is both the simplest to understand and, potentially, the most time-consuming. In this phase, you will take all of your work from the prior phases of development, and create meaningful and well-documented examples for your users. For this, I would highly recommend that you give your "beta" driver to a few of your existing customers and instruct them to stress test it. Let them tell you what's wrong and what could be better and work from there.

And remember: as a programmer, intimately familiar with the inner workings of your instrument, you are inherently biased. Take the time to remove that bias and think like your most basic users. All of the rest of your users will thank you for it.