Integrating VXI Plug&Play Drivers with PAWS

This application note gives information on the integration of VXI Plug&Play drivers into PAWS CEM 'C' coded drivers and on use of the 'Soft Panel' displays with these drivers. The first section will present an overview of PAWS CEM drivers. The second section will present an overview of VXI Plug&Play, and the third section will present an example of integrating a Plug&Play driver into a CEM and implementing or creating a 'Soft Panel'.

PAWS CEM Overview

The PAWS system allows for several methods for controlling ATE instruments. The simplest method involves CIIL instrumentation. The native output of the PAWS ATLAS compiler is MATE CIIL, so all you need to do for these types of instruments is write a resource static description. Most instruments don't have CIIL options, so you will probably need to have some other software driving the instruments. The remaining options are PAWS macrocode, which uses PAWS IDBDL (Instrument Database Description Language) a 'C' like language, and integrating external drivers using the PAWS WCEM (Windows CIIL Emulation) process.

In the WCEM process you will create a DLL that contains all the instrument driver code. The RTS will call functions in the DLL when an ATLAS statement is allocated to an instrument that is controlled by a CEM driver.

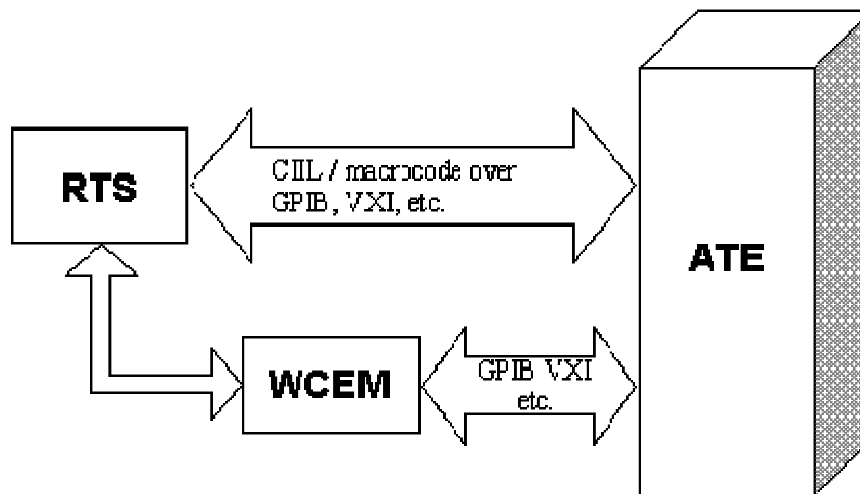

Figure 1.

Figure 1 shows the WCEM DLL communicating directly to the ATE. However when you build the CEM DLL you will include the driver software for the instruments you need to control. The WCEM may also contain references to VXI plug&play drivers, other drivers delivered by the manufacturer, or functions that you write yourself to control instruments. The WCEM may also reference other DLL's that contain driver code. These other DLL's will be loaded when the WCEM DLL is loaded by the RTS.

When referencing VXI plug&play drivers the system architecture of WCEM will include the following elements shown in figure 1b.

Figure 1b.

The WCEM will call functions in the VXI plug&play driver. That driver in turn calls the low-level functions in the VISA driver which interface with the hardware level. The WCEM is, when it is created, only an interface between the RTS and whatever instrument drivers are being referenced, in this case the VXI plug&play driver.

The VXI plug&play driver follows a defined architecture and provides a standardized interface for the application programmer. It insulates the programmer from the implementation details of low-level communications with the VXI instrument. The actual low-level communications are performed through the VISA (Virtual Instrument System Architecture) library.

The VISA library provides the low level functions that control opening and closing of sessions to VXI instruments, and communications with VXI devices. The VISA API (Application Programmers Interface) is defined in the VXI plug&play standards vpp4x which are available on the VXI plug&play consortium website at www.vxipnp.org.

The only part of the VXI plug&play standard you need to be concerned with is the actual plug&play driver for the instrument you are integrating. You need to have VISA installed on your system, and include the library when building your WCEM, but you will not be referencing any of the VISA function directly. You will be declaring variables of some VISA defined data types however, and will need to include the header `visatype.h` in your 'C' files. This will be discussed later in the example of integrating a plug&play driver.

The WCEM Wizard

The PAWS Developer's Studio uses a WCEM Wizard to create the functions that will be called for each instrument and ATLAS single action verb. When you select an instrument, FNC number, and single action verb, a list of ATLAS modifiers is presented as shown in figure 2.

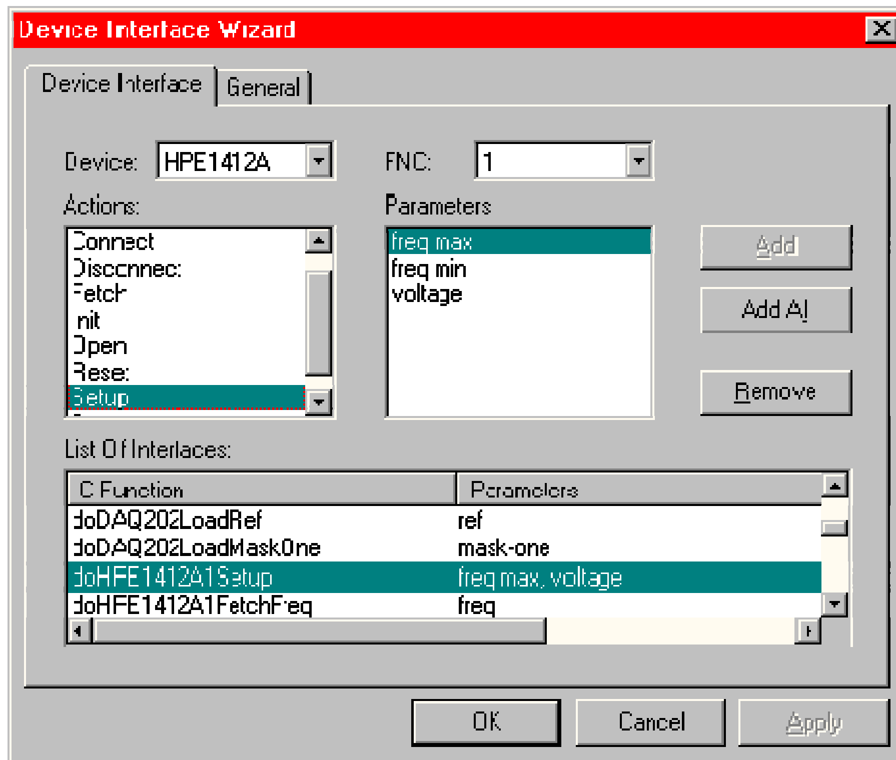

Figure 2.

From this list you will select which modifier values you want to pass to the WCEM function.

The WCEM Wizard will automatically select a function name for you, or you can type in your own. The Wizard maintains a list of functions that have been defined so you can keep track of your work.

Resource Static Description for WCEM Instruments

To create a WCEM using the PAWS Developer's Studio you need to start with a resource static description for your instrument. The static description will define the functionality of the instrument and gives the WCEM Wizard the keys to create the template for the WCEM. The static description will be the same as for any other type of driver, with a few additional rules.

First, you must use function numbers. The WCEM Wizard will create functions based on the instrument you assign in the static description, the single action verb being implemented, and the FNC number you assign.

Second, the FNC numbers must appear at the lowest functional level in the static description. This means if you have a `begin FNC = x` statement, you can't have any `begin/end` structures nested inside that structure. If you do, the WCEM Wizard won't be able to find them.

WCEM Files

When you use the WCEM Wizard a series of source files are created. The source files are the foundation for the WCEM.DLL that will be created when you build the CEM process. The files created by the Wizard are:

Wrapper.c	Contains dispatch functions that retrieves ATLAS data and passes it to the interface functions defined in the Wizard.
Key.h	Key table definitions used in Wrapper.c that define ATLAS nouns, modifiers, dimensions, etc.
Error.c	Error handling routines called from Wrapper.c
<device_name>.c	Instrument C file. Contains the interface functions defined in the Wizard. These functions are passed the ATLAS data from the dispatch routines in Wrapper.c. This is the only file you need to modify.
Ctlr.c	Contains the `DoIfc()` and `DoDcl()` functions which are used to setup the ATE to a quiescent state when the ATLAS program is loaded and reset the ATE when a program is unloaded.

In addition to these files you can add any other source files that you want to become part of the WCEM. These can be user defined C source files, plug&play drivers or other driver files. Pathnames for include files and libraries, and required libraries for the linker are specified in a WCEM options window.

These files will be discussed in greater detail later in this application note.

VXI plug&play Overview

The VXI plug&play specification provides a standardized approach for developing instrument drivers that provides the end user with familiar set of files and functions no matter who the manufacturer of the instrument was. Instrument drivers must be registered with the VXI plug&play consortium before they can be labeled with the VXI plug&play logo and certified to be in compliance with the specification.

The plug&play specification defines the interface that the application programmer will use to access the functions of the instrument. The overall system architecture of a system running plug&play is:

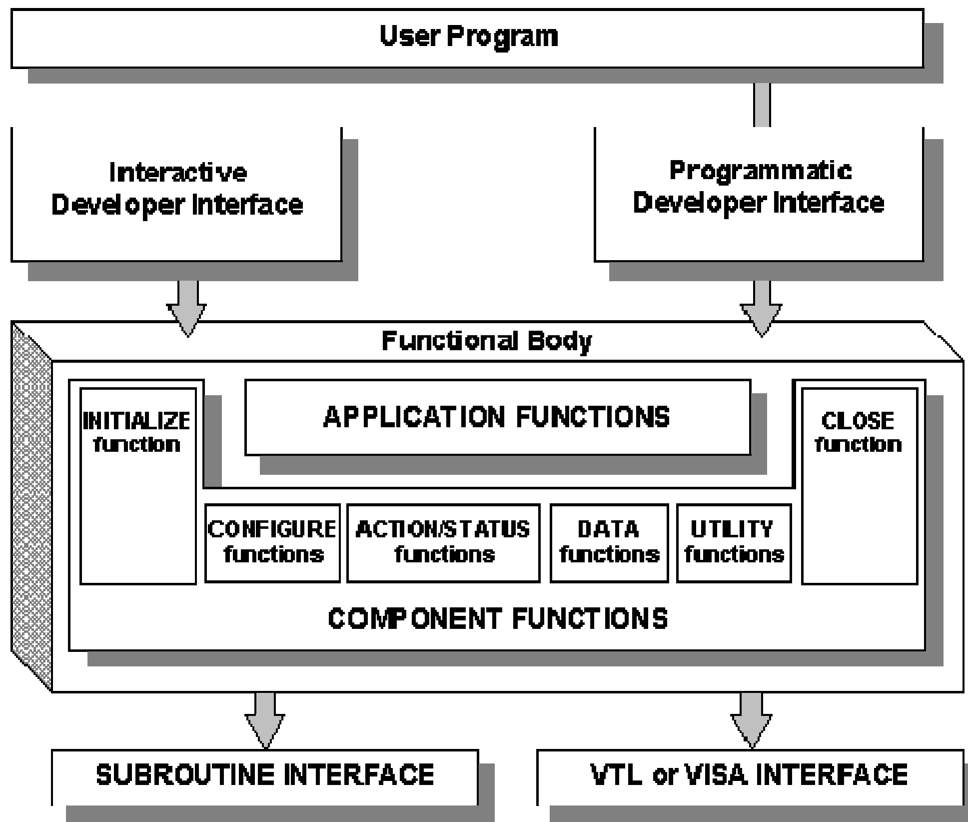

Figure 3.

The user program in the case of writing a WCEM is the <instrument name>.c, or ctrl.c file created by the Wizard. The access to the application functions and component functions of the plug&play driver is through the programmatic developer interface. This provides access to the functions of the driver through either Function Panels or through the C header file that contains the prototypes of the library functions in the driver. The contents of the function body of a plug&play driver are defined by the VXI plug&play specification, so all instruments will have an initialize function, a close function, and other functions that will be grouped as shown. These functions access the instrument through the VISA interface. This insulates the programmer from the low level implementation details of controlling the instrument.

You need only open a session to the instrument with the initialize function, program the device with one or more calls to the application functions, and when you are finished end the session to the instrument with a call to the close function.

Plug&Play Driver Compatibility issues

The contents of a plug&play driver are targeted to one or more ‘system frameworks’. The system frameworks define all the required elements for a VXI system under different operating systems. The system frameworks defined for VXI plug&play include WIN, WIN95, WINNT, SUN, and HP-UX. We will focus on the WIN95/WINNT frameworks as these are the systems that the PAWS Developer’s Studio runs under.

The system framework for WIN95 defines that a system will have a VXI framework

consisting of a VXI mainframe and a VXI slot 0 computer or resource manager. The WIN95 framework defines that a system will include:

- A computer that is 100% IBM compatible with
  - 486 33 Mhz or greater CPU with floating point
  - at least a 500 MB hard drive
  - a VGA monitor or higher
  - a 3.5-inch, 1.44-MB floppy disk drive
  - 16-MB or more of RAM
  - a Windows95 compatible mouse.
- The ability to control VXI message-based and register-based instruments.
- The VISA API dynamic linked library VISA32.DLL.

Most PC's or slot 0 computers easily meet these requirements. The VISA library is installed when you install the VISA driver on your system. National Instruments NI-VISA is an example of this driver. It typically installs under the directory C:\VXIPNP, which is the default for all your plug&play drivers. The file VISA32.DLL is installed under C:\WINDOWS\SYSTEM.

Plug&Play Driver Files

The plug&play driver will provide the following files as defined by the VXI plug&play specification:

  - ANSI C source code (.c, .h files)
  - A Windows 32-bit DLL (*_32.DLL file)
  - A Windows 32-bit import library (.lib file)
  - A function panel (.fp file)
  - A Visual Basic function declaration file (.bas file)
  - A VXI plug&play Knowledge Base text file (.kb file)
  - Driver documentation in a Windows help file (.hlp file)
  - An executable softpanel program.

The ANSI C source code provides source code for the driver functions, and the header file contains the prototypes for the functions. The header file will need to be included into WCEM instrument 'C' file. The driver source code is best left alone unless you know exactly what you want to modify.

The Windows 32-bit DLL contains the driver functions for the instrument. The file name will be the name of the instrument followed by '32.DLL' or '_32.DLL'.

The Windows 32-bit import library will need to be added to the library list for building the WCEM and its path name included in the library search paths.

The function panels are of particular interest if you are using LabWindows® /CVI. A function panel consists of a tree of functions that include the functions previously shown in the plug&play architecture. Each function call has a panel that shows it's prototype, provides a field for each argument, and provides help on each of the arguments. CVI contains a function panel editor which allows you to load the

instrument, open a function panel, and insert the call to the function directly into your 'C' file. This process will be discussed later in the application note.

The knowledge base file provides information about the instrument as it pertains to the VXI system. The WCEM process does not use the knowledge base.

The Windows help file is a useful reference that provides a description of each of the functions in the plug&play driver.

The softpanel is used initially to verify the functionality of the driver, and can also be used as a learning tool to teach instrument control concepts. When you first install your instrument and driver, run the softpanel program to verify that your instrument works. This also gives you confidence in the instrument library for the plug&play driver.

The softpanel supplied with the driver is typically much more complex than is required by an ATLAS runtime environment. The vendor-supplied softpanels are standalone executables that show all of the features of the instrument in multiple windows. A softpanel that would be used at runtime with the ATLAS RTS would typically function as an indicator, displaying the current mode and measurement. The functions in the DLL supplied with the driver do not reference the softpanel. If you want to have a softpanel as an indicator in your program, you probably want to design and implement your own using a tool like LabWindows/CVI.

When you install a plug&play driver the default directory is `C:\VXIPNP`. Under this director there will be a `Kbase` directory where all of the knowledge base files are stored, and a directory for each of the different frameworks that you install. If you installing the plug&play driver for the Hewlett Packard E1412A Digital Multimeter under the WIN95 framework they would be installed in the directory '`C:\VXIPNP\WIN95\HPE1412`'. Additionally under the `WIN95` directory would be the directories of `BIN`, `INCLUDE`, and `LIB`. The `BIN` directory contains the DLL files for each instrument under the framework. The `INCLUDE` directory contains all the header files for instruments under the framework, and the `LIB` directory contains subdirectories for each 'C' compiler supported which contain the driver source code specific to that compiler.

As you install more VXI plug&play drivers they will be added to the VXIPNP\WIN95 directory. This provides a common location and an organized structure for maintaining all of the plug&play driver source files, which also makes setting up the WCEM easier, as it reduces the number of paths that need to be added for include files and libraries.

Integrating a plug&play Driver

In this section we will demonstrate integrating a plug&play driver by using a simple example and a plug&play driver for the Hewlett Packard E1412A Digital Multimeter.

Required Software

Before attempting to integrate a plug&play driver you need the following:

- TYX PAWS Developer's Studio version 1.2.0 or later and Run-Time System.
- Microsoft Visual C++ version 4.0 or later
- National Instruments NI-VISA driver version 1.2 or later. (Downloadable from national instruments).
- The VXI framework required elements
- The WIN95/WINNT framework required elements
- The VXI Instrument and plug&play driver
- (optional) National Instruments LabWindows/CVI version 4.0 or greater.

LabWindows/CVI is optional but it makes the job of integrating plug&play drivers much easier as will be shown in this example.

Step One – Know your Instrument, Know your ATLAS

The first step in writing any instrument driver is to learn the capabilities of the instrument and how they relate to ATLAS. Remember that all calls to the plug&play functions will be in the context of an ATLAS signal oriented statement. You need to look at the instrument capabilities, what is needed to make a measurement, and then map those to the ATLAS single action verbs, nouns and modifiers that will be used in the test programs.

For simplicity we will implement only the DC voltage measurement features of the E1412A. The DMM is capable of measuring –300 vdc to +300 vdc in five ranges. We will see that the design of the functions for the E1412A allows you to specify a max voltage, and not worry about the range to place the instrument in.

The ATLAS nouns and modifiers that can reference the DC voltage measurement will include DC SIGNAL/VOLTAGE, AC SIGNAL / DC-OFFSET, SQUARE WAVE/DC-OFFSET, and any other noun that allows the DC-OFFSET to be measured.

The ATLAS statements that will be used to measure the dc voltages include:

```
MEASURE, (VOLTAGE INTO 'MVAL'), DC SIGNAL,

VOLTAGE RANGE 10 V TO 20 V,

CNX HI J1-1 LO J1-2 $

MEASURE, (DC-OFFSET INTO 'MVAL'), AC SIGNAL,

DC-OFFSET RANGE –10 V TO 10 V,

CNX HI J1-1 LO J1-2 $

MEASURE, (DC-OFFSET INTO 'MVAL'), SQUARE WAVE,

DC-OFFSET RANGE 0 V TO 5 V,
```

```
CNX HI J1-1 LO J1-2 $
```

Additional modifiers could be added to further describe the signal, such as the VOLTAGE and FREQ of the AC SIGNAL or SQUARE WAVE, but these are the basic ATLAS statements we would use to control the instrument. There are other nouns that support DC-OFFSET, but we will just use these for this example.

You could also write single action verbs to test the function of the driver. This depends on your specification for your ATE. If you require full single action verb control over all signals, then this will change how the driver is written. There are functions for the instrument that will configure the DMM and make the measurement in one call. But if you need to have full single action verb control (SETUP, CONNECT, CLOSE, INIT, FETCH, OPEN, DISCONNECT, RESET) then you will probably need to use lower level function calls to implement each single action ATLAS verb.

Like any project the better your specification is, the better your final product will be. If you have defined up front all the system requirements you will save yourself a lot of effort redesigning the driver later.

The Resource Static Description

The resource static description defines the functionality of the instrument for the resource allocation process. All instruments will be defined in terms of a resource name, a function description, ATLAS modifier ranging, and connection ports.

The resource name defines the name that the instrument driver (on the PAWS side) will be known by. This name will appear in the BusConfi file to assign a controller and logical address for the device, and it will also be used by the WCEM Wizard for the name of the C file the driver functions will be created in. So choose a name carefully, making sure that you don’t choose the same name as the plug&play driver uses, to avoid any conflicts.

In the resource static description for this device we will describe three separate functions; one for each noun/measured characteristic being implemented for this instrument. A simple static description would include:

```
begin DEV DMM using E1412A;

cnx hi E1412A-HI, lo E1412A-LO;

begin FNC = 1; ** sensor (voltage) dc signal

sensor (voltage) dc signal;

control voltage range –300.0 v to 300.0 v;

end;

begin FNC = 2; ** sensor (dc-offset) ac signal;

sensor (dc-offset) ac signal;
```

```
control

{

dc-offset range –300.0 v to 300.0 v;

voltage range 0.0 v to 300.0 v;

freq range 3.0 hz to 300.0 khz;

}

end;

begin FNC = 3; ** sensor (dc-offset) square wave;

sensor (dc-offset) square wave;

control

{

dc-offset range –300.0 v to 300.0 v;

voltage range 0.0 v to 300.0 v;

freq range 3.0 hz to 300.0 khz;

}

end;

end;
```

Note several things about this static description. First, the resource name is DMM. The using defines that this resource can only be allocated to one ATLAS statement at a time, and it also identifies the instrument that this static description is being written for. The name could also have been E1412. You need to make sure that you don't have any naming collisions. By naming this device DMM a 'C' interface file called 'DMM.C' will be created. If the device had been named 'HPE1412' a 'C' interface file called 'HPE1412.C' would have been created, which could have been a name conflict with the plug&play source files. You can decide on a naming convention that suits your project and style of programming.

Second, note that the FNC numbers are all assigned at the lowest functional level. There are no begin/end structures nested beneath them. This is important, as the WCEM Wizard will not find any FNC numbers that are not at the lowest functional level.

Once you complete the resource static description for the E1412A you can build the device file. After the device database has built with no errors you can execute the WCEM Wizard. This process will create the interface for the WCEM.DLL where you can add the calls to the plug&play driver.

Integrating VXI Plug&Play Drivers with PAWS

The WCEM Wizard

The WCEM Wizard helps you create the interface layer of the WCEM that will be used to call the instrument driver functions for all your plug&play controlled instruments, as well as other 'C' controlled devices.

To create the WCEM interface, first select the instrument, in this case DMM. Next select the FNC number you want to create interface functions for. In our static description FNC 1 is DC SIGNAL / VOLTAGE. Third you select the single action verb you want to implement. Once you select a single action verb a list of modifiers for that verb/FNC will appear. You create an interface function or add a modifier to an exiting function by clicking on the modifier, and then click 'Add'. If you want to add all the available modifiers for that verb/FNC, then click on

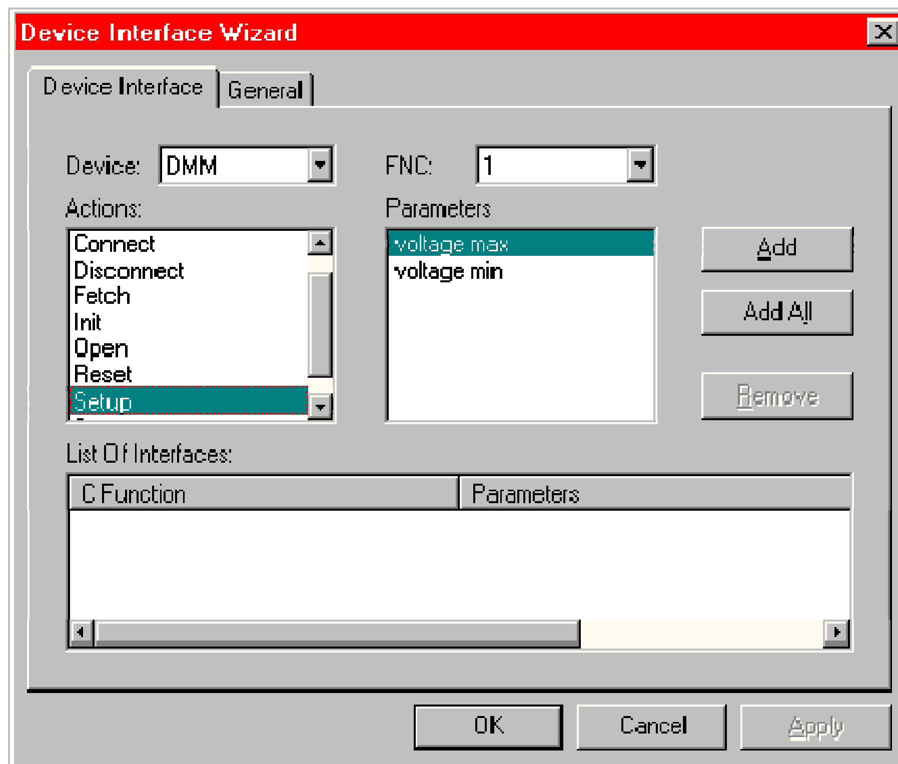

Figure 4.

'Add All'.

Note that the measured characteristic for a verb/FNC will show up as <modifier> max, and <modifier> min. This allows you to pass the range from the ATLAS sensor statement, or pass the driver function the maximum expected signal value.

The single action verbs available for analog sensor devices are (listed by function, not alphabetically):

Setup	The setup verb is the first step of all multiple action sensor statement. All modifiers are available in setup.
Connect	The switching action. This is called after setup, but the call should be to the switch driver, not the DMM.
Close	This function allows you to close any internal relays, or enable an output of a source signal.
Init	The init function triggers a measurement for a sensor statement.
Fetch	The fetch verb retrieves a measured value for an analog sensor statement.
Open	This function opens any internal relays or disables an output for analog source statements.
Disconnect	The switching action. This function opens the system relays. It should make a call to the switch driver if your switch is WCEM controlled.
Reset	This function returns the instrument to it's reset state. This function de-allocates the resource. After reset an instrument may be used by another ATLAS statement.
Status	Allows you to use the SetFault() macro to assert an interrupt back to the Device Database where it can be processed by a Device Database macro.

You are not required to implement all of the single action verbs. For a sensor device you only need to implement the SETUP, FETCH, and RESET actions. For an analog source instrument you are only required to implement SETUP and RESET. The other single action verbs give you greater control over your instrument. And if you don't implement an action, the ATLAS programmer can not use that single action verb in their ATLAS program.

Careful analysis of your system requirements will guide you in the right direction in deciding what functions to implement.

When you select to 'Add' a modifier when a function does not yet exist, a box will appear on screen prompting you to specify a name for the function or to accept the default name.

The default name is 'do[inst name][FNC number][single action verb]. You are free to enter any name you like in this field. Once you add the function it appear in the 'List of Interfaces' on the Wizard. You then step through adding interface functions for all the single action verbs you wish to implement, and then proceed

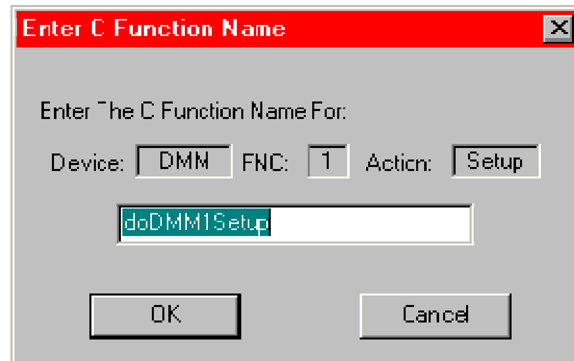

Figure 5.

to the next FNC number.

Figure 6 shows the first interface function, doDMM1Setup(), in the list of interfaces.

When you click on OK, the Wizard will create the source files Wrapper.c, Key.h,

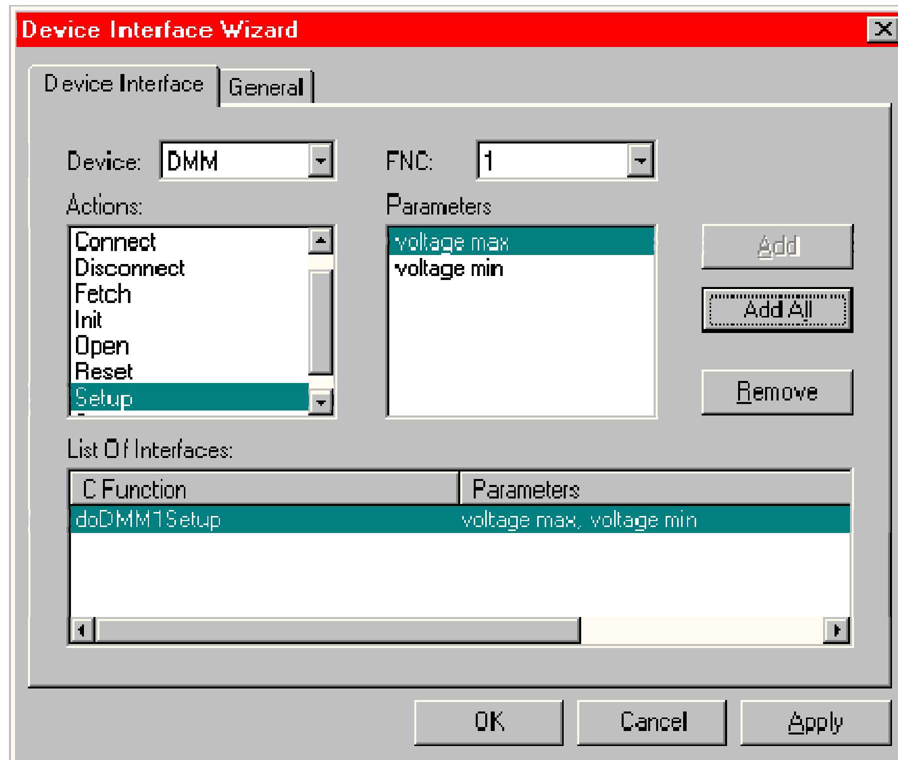

Figure 6.

Error.c, and DMM.c. They will appear in the PAWS Developer's Studio project workspace in the 'CEM Files' folder.

Adding General Functions

The general functions of interface clear and device clear are added from the 'General' tab of the WCEM Wizard. All you need to do is check the box next to the function you want to add, and either accept the default function name or type in your own function name. When you click on 'Apply' or 'OK' the file ctlr.c will be created and added to the project.

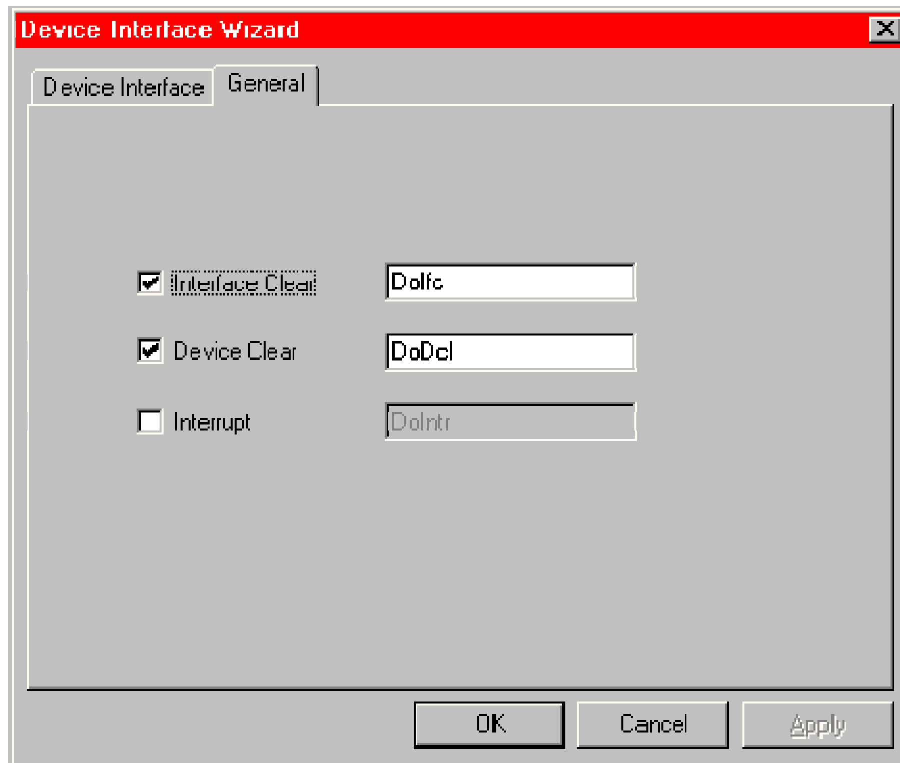

Figure 6b.

These functions are called any time an ATLAS program is loaded or unloaded. You will make calls to all the plug&play instruments in your system and verify that they return a good status, and place them in a ready state.

Setting up the WCEM Environment

Prior to building any of the files in the CEM you need to setup the WCEM environment. Selecting 'CEM' from the 'Options' menu accesses the WCEM options.

The first tab, 'Options', allows you to specify the output directory of the build process, any compiler options, and any linker options. Unless you have specific needs that require you to change or add to these lists, the defaults will be sufficient.

The 'Files' tab, as seen in figure 8, is used to setup the path names to search when

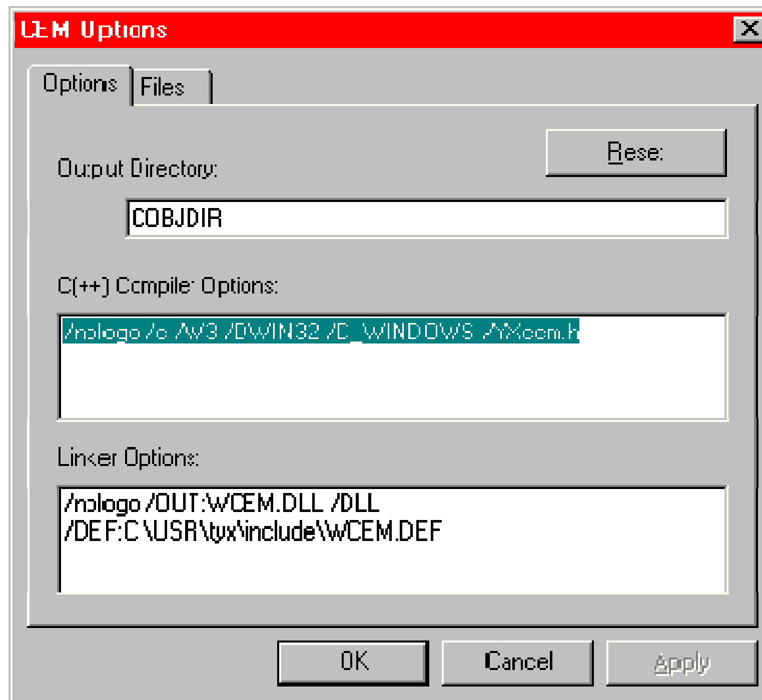

Figure 7.

locating include files and library files. You also specify the names of any libraries to be included during the link phase.

For VXI plug&play drivers there are two directories you need to add to this list, and several files. In the 'Include Path' list, add the directory 'c:\vxipnp\win95\include', and in the library file list, add 'c:\vxipnp\win95\<inst_directory>', which in this example is: 'c:\vxipnp\win95\hpe1412'. Remember that the 'win95' directory is only if you are using the WIN95 framework. If you are using a different framework, such as WINNT, examine your directory structure to be sure of the path name.

Unless you are using a different compiler and are sure about its usage, leave the 'Compiler' and 'Linker' fields alone. The 'Include Path' is where you will add the directories for include files, separated by semicolons. The same applies to the 'Library Path' field.

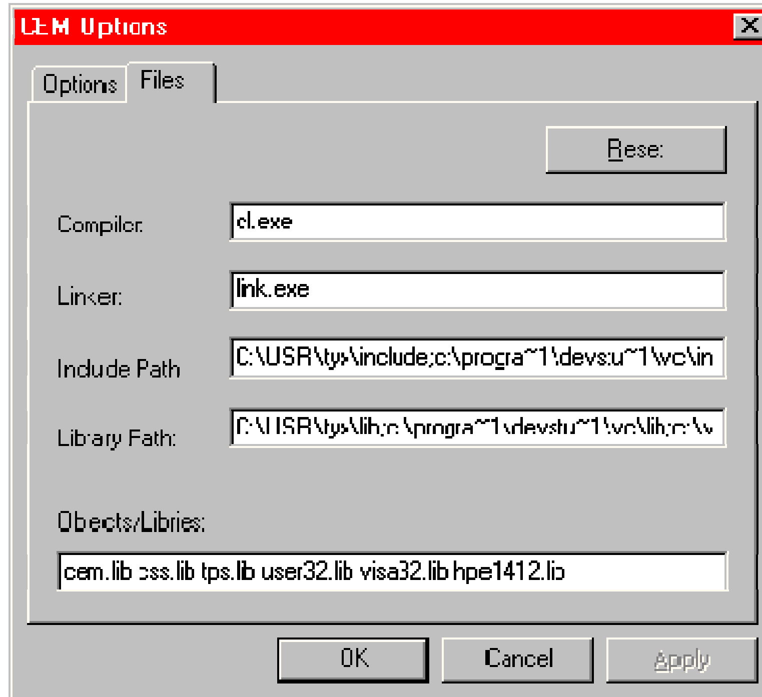

Figure 8.

The 'Objects/Libraries' field is where you will add any libraries required for linking. You must add the import library for the visa32.dll, `visa32.lib`, and the import library for your plug&play driver, in this case `hpe1412.lib`.

Adding Plug&Play calls to <instrument>.C

At this point we've done our homework and studied the instrument and understand how to program it, what ATLAS code will be used to call it, and have created a resource static description that accurately describes the nouns and modifiers that the instrument supports. Using the WCEM Wizard we created the interface to the instrument, and now we're ready to make calls to the plug&play driver to control the instrument.

To facilitate this process we will use LabWindows® /CVI. This tool allows you to load the plug&play drivers function panel, and insert the call directly into your 'C' file, as well as declare any variables required by the function call.

Starting CVI and Loading the Instrument

The first step in this process is to start CVI and create a new project workspace. Next, to load an instrument, click on 'Load' from the 'Instrument' menu. When the file selection menu appears change directory to where your plug&play driver is installed, in this case '`c:\vxipnp\win95\hpe1412`'. Click on the .FP file and 'OK', and the plug&play driver will now appear under the instrument menu as shown in figure

9.

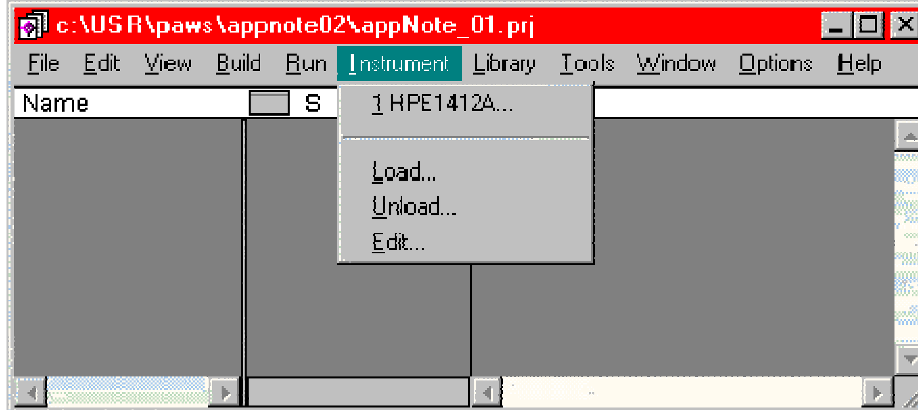

Figure 9.

When you click on the entry for your instrument, in our case the HPE1412, the function panel tree for the instruments plug&play driver will appear. This will be used when you open the 'C' file created by the PAWS WCEM Wizard with CVI to insert function calls into the interface file.

The function panel is displayed in a tree format, with the initialize and close functions on the top level, just like in the plug&play specification architecture, with additional categories containing functions as shown in figure 10.

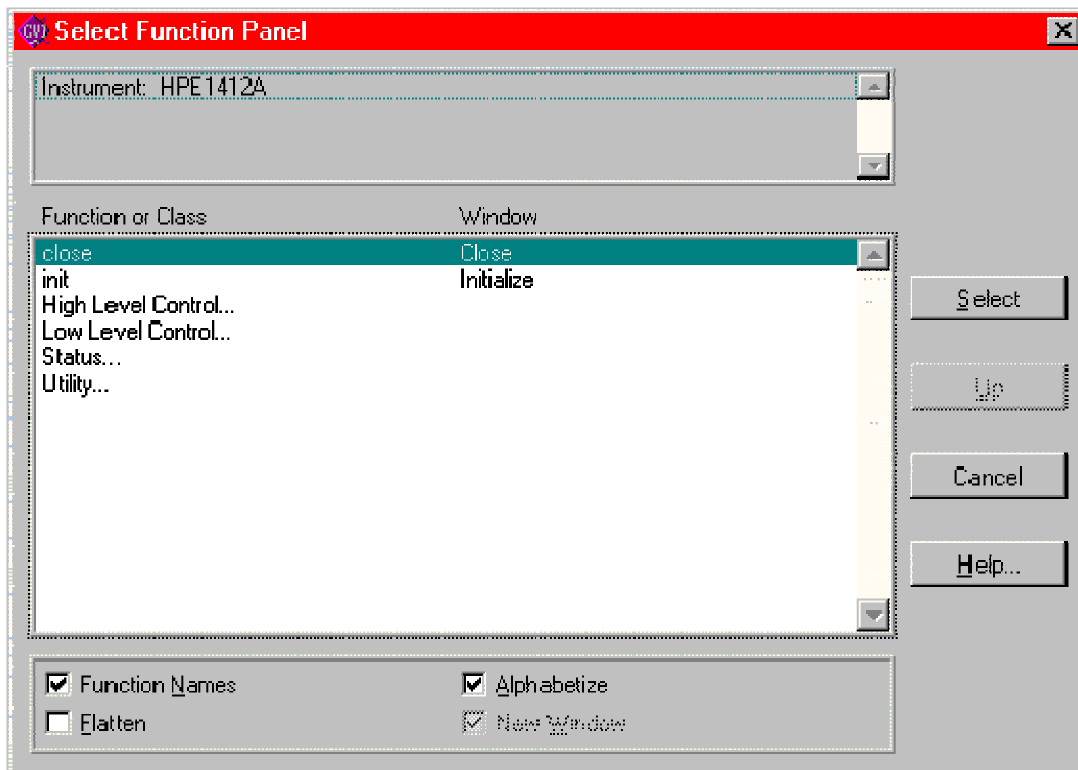

Figure 10.

Figure 10 shows the function categories of ‘High Level Control…’, ‘Low Level Control…’, ‘Status…’, and ‘Utility…’. The options for the function panel viewer allow you to specify a flat list, meaning all functions will be shown in the same level, and you can also alphabetize the list.

Editing <instrument>.C

The next step is to put add the instrument function calls. First, load the <instrument>.c file that was created by the WCEM Wizard into the CVI editor.

We need to add several `#include` statements to our ‘C’ file. First, add the header file for the plug&play driver, in this case ‘`hpe1412.h`’. You also need to add the visa data type header ‘`visatype.h`’. This contains the definitions of all the NI-VISA data types that are referenced in the plug&play drivers.

The programming technique used in this example for each WCEM interface function will be to:

- open a session to the instrument with the initialize function
- configure the instrument
- close the session to the instrument

To insert a function call in the CVI editor, place the cursor where you want the function call to go, and select the instrument from the ‘Instrument’ menu in the editor. The function panel tree will appear as previously seen in figure 10. The first step in to open a session to the instrument, so we will select the initialize function.

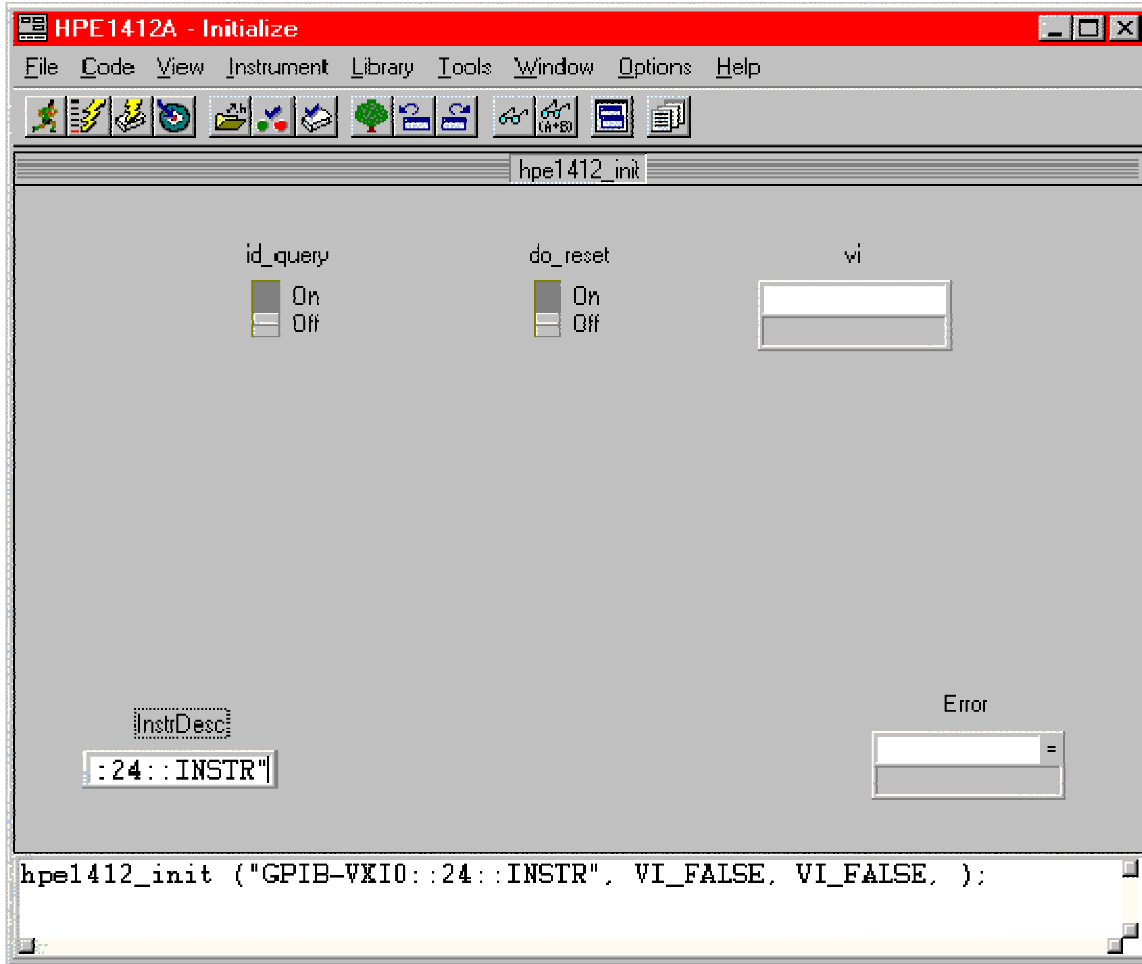

Figure 11.

The format of the function panel provides a window or a control for each parameter required. Right clicking on the body of the function panel provides overall help on the function, including a brief description and the function prototype. Right clicking on a field or control provides help on the field.

The parameters for the initialize function allow you to perform an ID query and reset the instrument. The parameter 'vi' returns a ViSession handle to the instrument, passed by reference. This handle is used by other function calls to specify the hpe1412. The instrument description describes the type of controller and the logical address. This field is dependent on the controller you are using, so check your documentation for the specifics on the handle.

When you need to declare a variable for an argument to a function, such as the vi handle, or a status variable, or can do it right from the function panel. If you place the cursor in a field, you can select 'Declare Variable' from the 'Code' menu. The 'Declare Variable' window will appear on screen.

Figure 12.

This feature will declare a variable of the correct type, and allows you to place the declaration at the top of the target file, or in the current function. In this example we are declaring a ViSession variable which will hold the handle to the hpe1412. As this will be referenced by all the functions it is placed at the top of the file as a static variable.

<1> c:\USR\paws\appnote01\DMM.c

File Edit View Build Run Instrument Library Tools Window Options Help

```
static ViStatus hpe1412Status;
static ViSession hpe1412Handle;
#include "com.h"
#include "key.h"
#include "hpe1412.h"     //hp e1412 plug&play driver include file
#include "visatype.h"    // visa datatypes include file

#define HPE1412_DESC "GPIB-VXI0::24::INSTR" //instr description for in

//BEGIN{DFW}.DMM.1.0
int doDMM1Setup (double MaxVOLT, double MinVOLT)
//END{DFW}
{
    hpe1412Status = hpe1412_init (HPE1412_DESC, VI_FALSE, VI_FALSE,
                                  &hpe1412Handle);

    return (int) 0;
}

//BEGIN{DFW}.VOLT.DMM.1.2
double doDMM1FetchVoltage (void)
//END{DFW}
```

16/97 1C Ins

Figure 13.

When you have declared all the variables and filled in all the fields on the function

variable declarations created by the function panel are inserted at the top of the file. The `#include` statements have been added for the required libraries. Also the string "GPIB-VXI0::24::INSTR" has been assigned as a `#define` constant. The string "GPIB-VXI0" implies this is a GPIB to VXI controller, number 0. The string "24" states that the Logical Unit Address of the DMM is 24, and the string "INSTR" states that this is an instrument. Again, this string is dependent on the type of controller, so check with your documentation.

Now we'll fill in the remaining implementation details for this interface function. We need to place the DMM in DC Voltage mode, and set the range to the correct value. Under the low level function/configuration functions we find a function called '`hpe1412_voltDcRang()`'. This function sets up the DMM in DC Voltage mode, and configures the range. On the function panel for this function the voltage range is by default set by a slider control. If you click on the control, then go to the options menu you will find a selection called 'Toggle Control Style'. This changes the control from a slider to a field where you can enter the variable name.

The setup interface function receives two arguments, the MaxVOLT and MinVOLT. These will be the high and low end of the VOLTAGE RANGE in the ATLAS statement. ATLAS determines the max and min values by value, not by magnitude, so we'll need to compare the absolute values of the two variables to make sure we pass the greater of the two. This will protect against an ATLAS statement with a voltage range field '`VOLTAGE RANGE –75.0 v to 1.0 V`'. In this case the max value would be -1, and the min value would be –75. This would be a problem if we only passed the MAX value to the hpe1412 function. You would set the instrument to a 3.0 v range, when the max expected signal value is 75.0 volts.

You may also want to check the return values of each of the plug&play calls to insure that no errors occur. An easy method is to define a common error handling function that can be called after any function returns a code other than VI_SUCCESS. In this example there is an external function defined called PlugNPlayError(). This function would be placed in the file Error.c, and could be called by any function where a plug&play driver returns an error code. You can decide what the best method for processing errors is in your system. You may want to halt the RTS and display an error message so that the ATE maintenance engineers can troubleshoot the failure.

After configuring the DMM we close the session to the device. We place a call to the close function for the hpe1412 at the end of our interface function, and return with a zero(0) status. If you return a negative value, the Wrapper.c function that dispatched the call to `doDMM1Setup()` will call a default error handling routine and report a bus error for the device DMM. You may wish to use this default error reporting mechanism and just return a –1 when an error is detected in an interface function.

<1> c:\USR\paws\appnote01\DMM.c

File Edit View Build Run Instrument Library Tools Window Options Help

```
 7  #define HPE1412_DESC "GPIB-VXI0::24::INSTR" //instr description for init
 8
 9  static ViStatus hpe1412Status;
10  static ViSession hpe1412Handle;
11
12  extern void PlugNPlayError(char *function, ViStatus code);
13
14  //BEGIN{DFW}:DMM:1 0
15  int doDMM1Setup (double MaxVOLT, double MinVOLT)
16  //END{DFW}
17  {
18      // open a session to the hpe1412
19      hpe1412Status = hpe1412_init (HPE1412_DESC, VI_FALSE, VI_FALSE,
20                                    &hpe1412Handle);
21      // call error handling routine in not successful
22      if(hpe1412Status != VI_SUCCESS)
23          PlugNPlayError("hpe1412_init()", hpe1412Status);
24
25      // configure dmm dc voltage range
26      if(abs(MaxVOLT) > abs(MinVOLT))
27          hpe1412Status = hpe1412_voltDcRang (hpe1412Handle, VI_FALSE,
28              MaxVOLT);
29      else
30          hpe1412Status = hpe1412_voltDcRang (hpe1412Handle, VI_FALSE,
31              MinVOLT);
32
33      // call error handling if not able to set range
34      if(hpe1412Status != VI_SUCCESS)
35          PlugNPlayError("hpe1412_init()", hpe1412Status);
36
37      // close session to hpe1412
38      hpe1412_close (hpe1412Handle);
39
40      return (int) 0;
41  }
42
```

37/120 32 C Ins

Figure 14.

The completed interface function is shown in figure 14.

The only time you will not return 0 from an interface function is the fetch functions. In the fetch functions the RTS expects the return code to be the number of values returned. For an analog sensor like the DMM the return code from the fetch function should be a 1, informing the RTS that one value was returned. If you return a zero, the RTS will assume that no data was returned.

Adding Virtual Panels to the WCEM

One of the features of plug&play you may want to take advantage of is a soft panel. This allows you to have a virtual instrument display on the display of your computer showing the current state and/or measurement for an instrument. All plug&play drivers are delivered with a softpanel executable, however this file is not suited to integration with the WCEM. The manufacturer supplied executable is normally a very complex set of panels that controls all the features of an instrument, when what you need to use with the PAWS Run-Time System is indicators. If a soft panel is

processing interrupts then the RTS will be in a wait state. So this example will demonstrate the development and integration of an indicator panel for the HPE1412 DMM.

This example will show the use of LabWindows® /CVI to develop the soft panel, and will create a DLL that displays the function panel and provides an interface to update the display.

Softpanels are accessed in PAWS by creating 'dummy' instruments in the device database. Using the WCEM Wizard you will create a setup function for the dummy instrument. In the device database You will write a macro that executes a 'setdevice()' command for the dummy instrument, and then sends a CIIL string over the GPIB bus. Because the instrument is controlled by WCEM, the RTS will then execute the setup function for the dummy instrument. This setup function will activate the soft panel, and then other functions in the driver can update the panel.

Creating the Soft Panel DLL

For the purposes of this application note it is assumed that the engineer has some experience with LabWindows® /CVI and can create function panels and DLL's. This example will only present a brief overview of the process.

When you create the soft panel you don't need to create an exit button as the panel will never be processing interrupts. The panel will be displayed with a call to `InitUIForDLL()`, and will be removed with the call to `DiscardUIObjectsForDLL()`when the DLLMain is called for ProcessDetach.

For this indicator three functions were created; `hpe1412DispPanel()` which calls `InitUIForDLL()` and displays the panel, `hpe1412SetMode()` which updates the mode ring control on the panel, and `hpe1412DisplayMval()` which displays the measured value.

The soft panel that was created for this example is very simple, as shown in figure15.

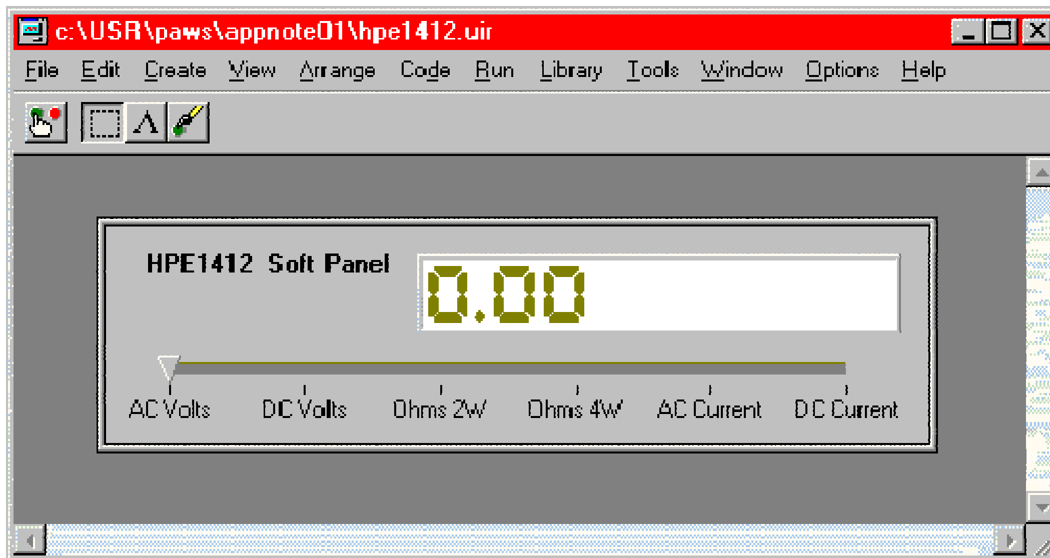

Figure 15.

There are no callbacks associated with this panel. It's sole purpose is to display the current mode and measured value of the DMM. You can create a panel to suit exactly what your needs are. You may opt for a small indicator only showing the measured value, or you may decide that you need a more complex panel that shows more information about the DMM.

The 'C' code generated by CVI, plus the functions added to control the panel take up only about 70 lines. In addition to the default code generated by the UIR editor, the following functions were added.

```
void hpe1412DispPanel()

{

InitUIForDLL();

}

void hpe1412SetMode(int mode)

{

SetCtrlVal (panelHandle, PANEL_DMM_MODE, mode);

}

void hpe1412DisplayMval(double val)

{

SetCtrlVal (panelHandle, PANEL_MEASUREMENT, val);
```

```
}
```

The following header file was created for the DLL and needs to be included in your 'C' file were the soft panel will be controlled:

```
#define ACV 0

#define DCV 1

#define OHM2 2

#define OHM4 3

#define ACA 4

#define DCA 5

void hpe1412SetMode(int mode);

void hpe1412DisplayMval(double val);

void hpe1412DispPanel(void);
```

In addition you will also need to include the export library for the DLL functions, hpe1412vpanel.lib, in the WCEM options window.

Setting up the Device Database for Calling Soft Panels

The Device Database is the starting point on the PAWS side when adding a soft panel. A dummy device for the soft panel needs to be added, plus a macro that will be called from the ATLAS program to activate the soft panel. If you want the soft panel loaded automatically, then you can skip this section and add a call to InitUIForDll() in the PROCESS_ATTACH section of the DLLMain(). This will automatically start the panel when the DLL is loaded.

In the Device Database, as a macro is needed to start the soft panel, you will need to include the Built In Function list. The code required to start a soft panel is:

```
def, mac, StartDmmPanel();

{

Dsp("StartVI..\n");

setdevice(<DMMPANEL>);
```

```
Talk("FNC DCS VOLT :CH53\r\n");

}

begin DEV DMMPANEL;

source dc signal;

control voltage -999.0 v;

end;
```

The macro `StartDmmPanel()` will be called to display the soft panel if the user wants to see the DMM status and measured values. The macro `setdevice(<DMMPANEL>);` sets the current device to our dummy instrument. The `Talk()` statement tries to send the CIIL string "FNC DCS VOLT :CH53\r\" to the DMMPANEL. Because the dummy instrument is controlled by the WCEM as defined in the BusConfi file (discussed later), the RTS makes a call to the setup function for DMMPANEL.

The static description describes a source for dc signal, but this is just to keep the compiler happy. The statement "control voltage –999.0 v" guarantees that no signals will be allocated to this device.

After entering this code into the Device Database, run the build process. After the Device Database compiles run the WCEM Wizard and create a setup function for the dummy instrument DMMPANEL. This is the only function required. In the 'C' file that is created, DMMPANEL.C, the only call you need to make in the setup function is to the `hpe1412DispPanel()` function. I addition you can create a function that returns TRUE if the panel is active.

```
//DMMPANEL.C

#include "cem.h"

#include "key.h"

#include "hpe1412vpanel.h"

static unsigned short DMM_PANEL_STATE = 0;

//BEGIN{DFW}:DMMPANEL:0:0

int doDMMDisplayPanel (double VOLT)

//END{DFW}

{

hpe1412DispPanel();

DMM_PANEL_STATE = 1;

return (int) 0;
```

```
}

unsigned short IsDmmPanelActive()

{

return DMM_PANEL_STATE;

}
```

In the file DMM.C any function that would update the soft panel, it can first call the function `IsDmmPanelActive()` which returns non-zero if the panel is active. If the panel is active they can then call one of the two functions to update the mode or the measured value. Don't forget to add the prototype for this function to a header file or add the line:

```
extern unsigned short IsDmmPanelActive(void);
```

to the file DMM.C.

ATLAS Code to activate a Soft Panel

In the ATLAS program all we have to do is write a DEFINE statement for the macro to activate the soft panel, and then write a PERFORM statement if the operator wants to view the panel.

```
.

.

010000 DEFINE, MACRO, 'StartDmmPanel' $

99 END, 'StartDmmPanel' $

.

.

101000 OUTPUT, C'DO YOU WANT DMM SOFT PANEL DISPLAYED?',

C'ENTER YES/NO ' $

05 INPUT, GO-NOGO $

10 IF, GO, THEN $

         1. PERFORM, 'StartDmmPanel' $

20 END, IF $

.

.
```

The DEFINE, MACRO... statement is a PAWS extension to the ATLAS language which allows you to call Device Database macros from the ATLAS code. When using this feature remember the macro name is case sensitive, and must be referenced exactly as it was defined.

Adding WCEM Devices to the BusConfi File

The PAWS Bus Configuration file, BusConfi, identifies all the instruments in the ATE, and identifies which bus they are on, plus the talk and listen addresses for each device. All instruments that are controlled by the WCEM will be assigned to a bus called "Channel".

```
; IEEE-488 Bus Configuration File -

"IEEE-488 Bus" 1 MLA 30 MTA 30 gpib0

"Channel" 2 MLA 50 MTA 50

DCP BUS 1 MLA 1 MTA 1

ACP BUS 1 MLA 2 MTA 2

AFG BUS 1 MLA 4 MTA 4

DWG BUS 1 MLA 10 MTA 10

DMM BUS 2 MLA 24 MTA 24

DMMPANEL BUS 2 MLA 99 MTA 99
```

This example has two busses, "IEEE-488 Bus" and the "Channel". The GPIB controller is gpib0 at address 30, and the WCEM is shown with address 50. The instruments DMM and DMMPANEL are shown on BUS 2, which is the WCEM process controller. Add all your WCEM instruments under the "Channel" controller. This tells the RTS that the instrument functions are controlled by WCEM. Remember that the name in BusConfi must be exactly as define in the Device Database.

Building the WCEM

After implementing all the functions in your WCEM interface C files you are ready to build the WCEM.DLL.

Before you can build the WCEM you need to have an ATLAS program in the PAWS project, a switch database, and an ITA database (you already have a device database or you couldn’t created the WCEM files).

You can compile each of the ‘C’ files individually to verify they are syntactically correct. To compile a ‘C’ file, double click on the file in the PAWS project workspace. This will open an edit window with the ‘C’ file. Click on the compile button and the PAWS Developer’s Studio will invoke Microsoft Visual C++ and post the compile results to the error window of the PAWS Studio.

When you click on the 'Build Project' all files that are not current will be built in the following order:

Compile, Link, and Flow the ATLAS program

Build the Device Database

Build the Switch Database

Build the ITA Database

Resource Allocate the ATLAS program

Compile all CEM source files

Link CEM files and build WCEM.DLL, WCEM.LIB, and WCEM.EXP

If there are errors in any of the 'C' files, or unresolved symbols in the link process the errors will show up in the error window of the PAWS Developers Studio. If you encounter errors in your source code correct them and rebuild the project. If you get 'file not found' errors it indicates that Visual C++ can't find a header file or library file. This means you will have to add a directory to the 'Include Path' or 'Library Path' fields of the WCEM setup.

A successful build of the WCEM will yield an error report as seen in figure 15.

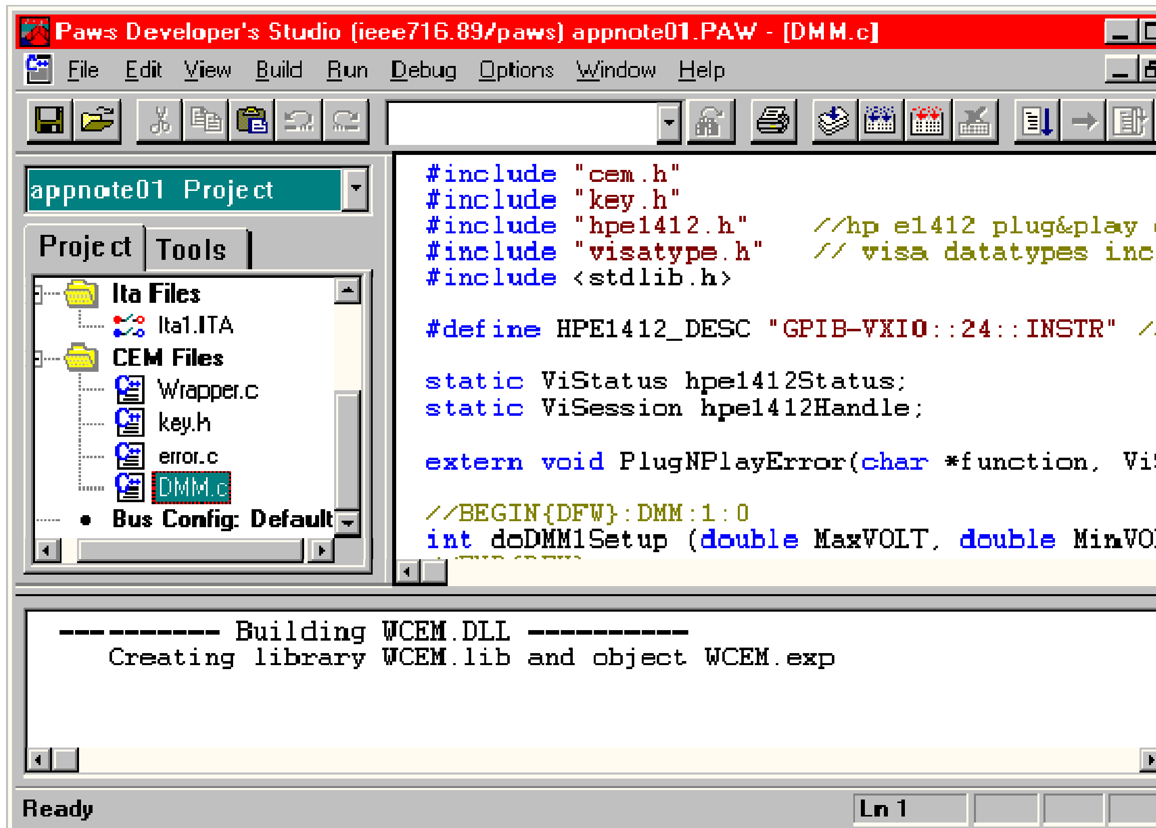

Figure 16.

Once the WCEM has built correctly the only errors you may encounter are run-time errors involving instrument status, bugs in your code, or addressing problems. If you ran the virtual instrument panel executable you would have verified the functionality of the plug&play functions and the device addressing.

Proper evaluation of the return values from the plug&play functions, use of the `SetFault()` macro to assert interrupts in the Device Database, and logging of errors through the functions in Error.c will allow you trap most of the run-time errors that can occur when using a WCEM. Write a comprehensive ATLAS test program that will exercise all of your driver functions and you will be able to trap the majority of problems in the integration phase.

For more on VXI plug&play…

For more information on VXI plug&play drivers you can try the VXI plug&play consortium website at www.vxipnp.org. All of the specification documents are downloadable as MS Word documents or Adobe Acrobat .PDF files. You can also find tutorials and papers on VXI issues.