ApplicationNOTE

GAMRY INSTRUMENTS

Customizing Electrochemical Experiments with the Explain™ Scripting Language

Introduction

Gamry Instruments has long given you the ability to modify your standard experiments as needed, or even develop your own experiments. All of our experimental scripts are written using an open-source scripting language, developed by Gamry, called Explain™.

The following discussion introduces the major ideas used in the Explain scripting language. The intent is to enable you to customize Explain scripts for your unique purposes. This note is not intended to serve as a complete reference for the Explain language. The On-line Help in the Gamry Framework™ software provides reference material both on the structure of the Explain scripting language and the functions supported within Explain. Also see the application note "Programming Reference for the Explain Scripting Language".

A few common customizations are detailed in our note "Commonly Requested Changes to Explain™ Scripts." You may find that the changes you need are described in that document. You can download all of these application notes from www.gamry.com/application-notes/.

Data-acquisition, of course, is only one part of the experiment. Data-analysis is also important. Gamry uses the Echem Analyst™ software platform to perform the analysis portion of the experiment. The Echem Analyst software consists of a number of scripts written in Microsoft® Visual Basic® for Applications (VBA). Like Explain scripts, these VBA scripts can be opened and modified for special applications.

The Three Levels of the Gamry Framework™ Software

The Gamry Framework™ environment contains three levels of programming.

1. Application code
    - Application code is responsible for generating the interface you see upon starting Framework. It also controls generation of the graphs during data-acquisition (but not the data displayed on the graphs).
2. Explain™ Scripts
    - Selecting an experiment from the drop-down **Experiment** menu in the Gamry Framework environment executes the corresponding script.
    - The Explain script creates dialog boxes, configures the potentiostat hardware, acquires the data, displays them in a graph on the screen, and saves the data to a file.
3. Compiled Code
    - Compiled code implements the Gamry application and many of the objects that are used by Explain language.
    - You have no access to this level. However, Gamry can still customize applications that require programming at this level.

What is an Explain™ Script and What are Object-oriented Languages?

The Explain™ language is a scripted, sequential, object-oriented language. To a non-programmer that statement makes little or no sense. The following section is an introduction to object-oriented languages.

Explain is a scripted language because Explain programs are composed of text characters that can be edited with a text-editor program. The Gamry Framework includes its own editor that you should use to edit Explain scripts. Select **File Open** from the main menu in the Gamry Framework and a dialog box called **Open a script for editing** appears (Figure 1). The files listed in that window with the extension `*.exp` are editable Explain scripts.

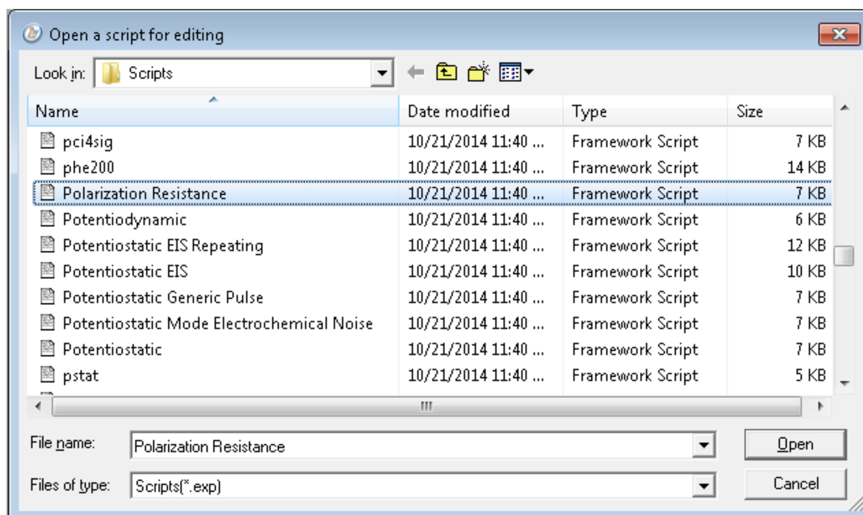

Figure 1. Selecting an Explain™ script for editing.

> **Functions in the Explain™ Language**
> Explain™ is a procedural language based on function calls and sequential execution of statements—similar to FORTRAN, Pascal and C. Explain, like common procedural languages, uses functions to manage execution flow. A function is a named part of an Explain script that can be invoked from other parts of the script as often as needed. As in most modern functional languages, each function has a name, an argument list, and a list of statements to be executed in sequence. Not all functions are Explain language functions contained within the script itself, Explain also includes library functions written by Gamry Instruments in compiled C.
>
> Explain follows the programming doctrine of an object-oriented language. Object-oriented languages have become quite popular (for example Visual Basic® and Java™). Object-oriented languages have their own terminology, referring to class and object, that can be confusing.

Explain scripts are sequential because they are executed from the beginning of the text file to the end. This allows you to easily follow the flow of the program. The exception to this rule is when a call is made to a function (See box: Functions in the Explain™ Language). When Explain encounters an unknown function it automatically searches the script for that function. The function is commonly set aside from the main body of the script by placing it near the end.

Object-oriented languages are built around "classes of objects". Classes may be considered collections of physical items that share characteristics. Examples of classes are the different shapes, `CIRCLE`, `SQUARE`, and `TRIANGLE` (by convention, classes are in uppercase in Explain). Each is a different class of objects. All objects that are a member of the `CIRCLE` class know that they are round; an object that is a member of the `SQUARE` class knows that it has four sides of equal length that are at right angles.

An object created from a class is called an "instance" of that class. For example, the following Explain code might create an instance of the `CIRCLE` class.

```
Shape1 = CIRCLE.New (Tag, X-coordinate, Y-coordinate, Radius, Color)
```

This new object is called `Shape1`. It is an instance of a `CIRCLE` class.

The most important idea in object-oriented languages is that objects combine both data structure and behavior into a single entity. An example is the `CIRCLE` class we previously discussed. The object has attributes such as a tag, a position, a radius, and a color.

Objects created from specific classes also know how to perform actions. Placing a period between the object and the action designates these actions. The following code is an example of how to create a new object of the class `CIRCLE`.

```
Shape1 = CIRCLE.New ("My Circle", 2, 5, 10, Blue)
```

The object called `Shape1` is created with the tag of `My Circle`, it knows that it is located at 2, 5 on the screen, it has a radius of 10, and the circle is drawn in blue.

In Explain, objects must be created before they can be used. That is why most Explain scripts contain a large number of object declarations using the `.New` function near the beginning of the script.

Once objects have been created they can perform certain tasks. For example now that we have created the object `Shape1` we could use `CIRCLE.Draw` to draw our circle on the screen. The following code would actually draw the circle on the screen.

```
Shape1.Draw()
```

Notice that because this object was previously defined it knows the tag, coordinate, radius, and color information. Because the object contains the data about its own properties, and knowledge of how to perform actions, complicated sequences can be easily condensed. For example if the `.Move` function was defined to move an object from one location to another on the screen the following code would erase the circle `Shape1`, and redraw it at the origin of our *x*-*y* graph.

```
Shape1.Move (0,0)
```

Another advantage is that there can be multiple instances of the same class. An unlimited number of instances of the `CIRCLE` class with different names could be created. Each circle would contain its own information regarding tag, coordinate, radius, and color that could be used when those objects performed actions.

Layout of a Typical Explain™ Script

The following discussion will use the Explain™ script for the corrosion test Polarization Resistance (`Polarization Resistance.exp`) as an example of the flow in a typical Explain script. Specific examples from the script will be shown and described. The script in its entirety can be found in Appendix A.

All Explain scripts begin with a brief description of what the script does. These "comment lines" use the semicolon (`;`) as the identifier that tell the Explain interpreter to ignore the rest of the line. For example, the following three lines are the first three lines of the script.

```
; Explain Script for Polarization Resistance Experiment
; Copyright (c) Gamry Instruments
; Version 6
```

There are a number of lines of code in Appendix A that contain a semicolon near the middle, or end, of a line of code. Semicolons do not need to be at the beginning of a line. The Explain interpreter treats the remainder of a line of code as a comment once it reaches a semicolon.

The next section is the "included" files. These statements allow functions to be added to a script by "including" other complete Explain scripts. The function of include statements is to allow commonly used pieces of code to be collected in a few places, and then be called when scripts are executed. In the polarization resistance experiment, the following files are included:

```
include "explain6.exp"
include "DC105.exp"
```

The `explain6.exp` script is contained in almost all other scripts, for it contains a list of "includes" that add functions to the Gamry Framework. Some more common Explain scripts are listed in Table 1.

Explain Script	Function
`Common.exp` `Common Functions.exp`	Library routines for functions and objects.

`signal.exp`
`curve.exp`
`pstat.exp`
	Information regarding specific potentiostat hardware control

`math.exp` `stats.exp`	Mathematical routines.
`Explain.exp`	Master include file: Includes all of the above files.
`Muxlib.exp`	Driver information for peripheral instruments, e.g., multiplexers and temperature controllers.

Table 1. Commonly-used Explain "include" files

Following the included files is the function `Main ()` declaration. This is a very important line, for the Gamry

Framework looks for function `Main ()` to indicate the beginning of an Explain script.

> **Important:** In Explain there are no BEGIN or END statements. Explain uses indentation (tabs) to indicate blocks of statements. The `Polarization Resistance.exp` script has all of the statements left justified until after the function `Main ()` declaration. The lines of code that are indented indicate that they are the block of code to be executed when the function `Main ()` is called. The majority of the sample script is part of the function `Main ()`.

In the layout of an Explain script, the object definition section begins immediately following the function `Main()`. This is a series of `CLASS.New` functions where new objects are created and default information is placed in these objects. This is identical to our `CIRCLE.New` example earlier. The script must create all objects needed for a specific experiment.

```
	Title = LABEL.New ("TITLE", 80, "Polarization Resistance", "Test Identifier")
	Notes = NOTES.New ("NOTES", 400, NIL, "&Notes...")
	Output  = OUTPUT.New ("OUTPUT", "POLRES.DTA","Output &File")
	Area    = QUANT.New ("AREA", 1.0, "Sample &Area (cm²)")
```

`LABEL`, `NOTES`, `OUTPUT`, and `QUANT` are just a few examples of classes of objects that can be defined. See the Gamry Framework On-line Help for definitions of each of these classes.

Following the creation of the needed objects, the script displays a setup window on the computer screen. This is done using the function `Setup ("Title", Item1, Item2, Item3, ....)`. The following code displays the setup box in the `Polarization Resistance.exp` script:

```
	result = Setup ("Polarization Resistance"
&		,PstatSelect.Selector (SELECTOR_ASTERISK)
&		,Title
&		,Output
&		,Notes
&		,VInit
&		,VFinal
&		,Scan
&		,Sample
&		,Area
&		,Density
&		,Equiv
&		,BetaA
&		,BetaC
&		,Condit
&		,Delay
&		,IRComp
&		)
```

The ampersand (`&`) symbol is used to disregard indentations. Program statements are generally one line long, but due to the limitations of the computer display they may extend beyond the edge of the screen. To simplify following the code, visually long program statements may be chopped into pieces displayed in sequential lines. When this is done the "`&`" allows the Explain interpreter to ignore any indentations on these segments of code.

After you have entered the parameters for the experiment and clicked the OK button, the setup window closes and the script creates a `Pstat` object using the information from the setup box.

```
	Pstat = PstatSelect.CreatePstat ("PSTAT", "PstatClass")
```

Different models of potentiostats can be installed in the same computer, so it is important to create the Pstat object containing information about the potentiostat immediately after the setup box is closed.

If any error-checking needs to be performed on the information entered in the setup box, it is done after the setup box closes. In general, this includes checking the number of data points to be acquired to see if they exceed the software limitations, checking to see if the data file already exists, and checking to see if any other script is using the potentiostat. If everything is deemed okay, the script exits the loop around that setup function.

```
Stdout ("Experimental Parameters")
Stdout ("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^")
Stdout ("Potentiostat: ", Config (GAMRYINI, Pstat.Section (), "Label"))
Stdout ("Start Time:   ", TimeStamp ())
Stdout ("")
Stdout ("Initial Voltage:   ", VInit.Show ())
Stdout ("Final Voltage:   ", VFinal.Show ())
Stdout ("Scan Rate:   ", Scan.Value (), " mV/s")
Stdout ("Sample Period:   ", Sample.Value (), " s")
Stdout ("")
Stdout ("Beta A:   ", BetaA.Value (), " V/Decade")
Stdout ("Beta C:   ", BetaC.Value (), " V/Decade")
Stdout ("")
Stdout ("Specimen Area:   ", Area.Value (), " cm^2")
Stdout ("Specimen Density:   ", Density.Value (), " g/cm^3")
Stdout ("Specimen Equivalent Weight:   ", Equiv.Value (), " ")
Stdout ("")
if (Condit.Check ())
       Stdout ("Conditioning Time:   ", Condit.V1 (), " s @", Condit.V2 (), " V")
if (Delay.Check ())
       Stdout ("Delay Time:   ", Delay.V1 (), " s")
if (IRComp.Value ())
       Stdout ("IR Correction:   On")
else
       Stdout ("IR Correction:   Off")
```

Figure 3. Explain script that produced the STDOUT window shown in Fig. 2.

The Explain script then writes the experimental parameters to the **STDOUT** (StanDard OUTput) window shown in Figure 2. Gain access to the **STDOUT** window during an experiment to review the setup parameters by selecting it from the curve drop-down list (`F5`) as shown in Figure 2.

The function `Stdout()` writes information to the **STDOUT** window. The value of objects, such as those in the setup box, can be written to the **STDOUT** window using `object.Show()` or `object.Value()` statements. A combination of text strings in quotations `and object.Show()` or `object.Value()` can be used to create complicated statements in the **STDOUT** window. The code shown in Figure 3 was used to produce the **STDOUT** window shown in Figure 2.

Figure 2. An Example of the STDOUT Window.

After the **STDOUT** window is created, the experimental parameters are saved as a header in the data file. This is performed using the `Printl()` function. It is very similar to the function `StdOut()` except that the information is written to the data file. Most setup objects have a `Printl()` instance function; this function knows how to print the object's tag and other information to the data file. The following is an example of `Printl()` functions from the polarization resistance script and the corresponding output. In some of the calls to the `Printl` function, `\t` is used to insert a tab into the file.

After the **STDOUT** and data file contain the setup parameters, the script then controls the potentiostat and performs the experiment. Before the actual polarization resistance scan starts, the script checks to see if the conditioning toggle box is selected and performs a conditioning step if it was selected. The following code checks the state of the toggle and

performs the conditioning. This condition function is available because it is defined in `DC105.exp` in beginning of this script.

```
      if (Condit.Check ())
            if (Condition (Pstat, Condit.V1 (), Condit.V2 ()
&                 ,IRComp.Value (),0.1*Area.Value ()) eq FALSE)
            return
```

This Explain Code	"Prints This to the Data file"
`Printl ("POLRES")`	`POLRES`
`Title.Printl ()`	`TITLE Polarization Resistance`
`Printl ("DATE\t", DateStamp ())`	`DATE 11/20/2000`
`Printl ("TIME\t", TimeStamp ())`	`TIME 0:01:55`
`Notes.Printl ()`	`NOTES 1`
`Pstat.Printl ()`	`PC4 Pstat0`
`VInit.Printl ()`	`VINIT -2.00000E-002 T`
`VFinal.Printl ()`	`VFINAL 2.00000E-002 T`
`Scan.Printl ()`	`SCANRATE 1.25000E-001`
`Sample.Printl ()`	`SAMPLETIME 2.00000E+000`
`Area.Printl ()`	`AREA 1.00000E+000`
`Density.Printl ()`	`DENSITY 7.87000E+000`
`Equiv.Printl ()`	`EQUIV 2.79200E+001`
`BetaA.Printl ()`	`BETAA 1.20000E-001`
`BetaC.Printl ()`	`BETAC 1.20000E-001`
`Printl ("CONDIT\t", Condit.Check ())`	`CONDIT F`
`Printl ("TCONDIT\t", Condit.V1 ())`	`TCONDIT 15`
`Printl ("ECONDIT\t", Condit.V2 ())`	`ECONDIT 0`
`Printl ("DELAY\t", Delay.Check ())`	`DELAY F`
`Printl ("TDELAY\t", Delay.V1 ())`	`TDELAY 300`
`Printl ("RDELAY\t", Delay.V2 ())`	`RDELAY 0.1`
`IRComp.Printl ()`	`IRCOMP F`

Table 1. Entering setup parameters into the datafile.

The script then checks the value of the open-circuit measurement toggle box and performs an open-circuit potential measurement for either the entered time and stability setting, or for 10 seconds. It then writes the value of the open-circuit potential to the data file. Again note that the OCDelay function is available because it is a function that was also included in `DC105.exp`.

```
      if (Delay.Check ())
            OCDelay (Pstat, Delay.V1 (), Delay.V2 () * 0.001)
      else
            OCDelay (Pstat, 10.0, NIL)
      Printl ("EOC\tQUANT\t", POTEN.Eoc (), "\tOpen Circuit (V)")
```

Executing the call to the `Cpiv` function performs the actual electrochemical experiment.

```
      ;  Run the curve
      Cpiv (Pstat, VInit.VsEref ()
&           ,VFinal.VsEref ()
&           ,Scan.Value ()*0.001
&           ,Sample.Value ()
&           ,IRComp.Value ()
&           ,EQDelay.Value ()
&           )
```

Because this is a function call, the Gamry Framework searches the script to see where the function `Cpiv ()` is defined and performs the actions in that block of code. The function `Cpiv` is defined near the end of the polarization resistance script. When the function call is executed, the information contained in a number of objects is "passed" to the `Cpiv`

function. These objects are listed in the parentheses following the `Cpiv` function call shown above. These objects are passed to the `Cpiv` object because function `Cpiv ()` is defined outside of the function `Main ()` portion of the script. Explain needs to know what information from the function `Main ()` must be known in the `Cpiv` function.

We now skip down to the function `Cpiv ()` section of the script to continue the sequence of this script. The first function in this section of code is the `InitializePstat()` function call. This requires a search of the script for the block of code in which the function `InitializePstat()` is created. This section of code immediately follows the block of code for the function `Cpiv`.

The `InitalizePstat` function contains a number of `Pstat.SetHardware` functions. These functions control the hardware settings on the `Pstat` object (which is the physical potentiostat that will perform the experiment this script is executing). For example, the code:

```
Pstat.SetCtrlMode (PstatMode)
```

sets the instrument in potentiostat control mode (all Gamry potentiostats can operate in either potentiostat, galvanostat, or zero resistance ammeter modes). The `InitalizePstat` function collects all of the hardware control settings in one location in the script, usually at the end, so that the hardware settings are clear. For a complete description of all hardware function parameters, see the On-line Help in the Gamry Framework environment.

Once the `InitializePstat` function is executed, the script returns to the `Cpiv` function and continues. The next step is to create a `Signal` object. `Signal` objects are a special class of objects in Explain. The physical parallel to the `SIGNAL` class would be the function generator. A `Signal` object creates the waveform that will be applied by the potentiostat to the electrochemical cell. In the `Polarization Resistance.exp` script the object `Signal` is created as an instance of the `VRAMP` (Voltage Ramp) class. The `VRAMP` class needs to know the initial potential (`Vinit`), final potential (`Vfinal`), scan rate, and sample time. With these parameters, the `Signal` object creates the discrete voltage values for the experiment. The object `Signal` is created with parameters by the following code:

```
Signal = VRAMP.New ("SIGNAL", Pstat, VInit, VFinal, ScanRate, SampleTime)
```

This is an example of an object created using information stored in other objects. Various `SIGNAL` classes exist for different types of waveforms that can be sent to the instrument. Table 3 is a list of some of the `SIGNAL` classes. Additional `SIGNAL` classes may be added as Gamry adds new applications to our software suite.

`SIGNAL` Class	**Description**
`V(I)RAMP`	Ramp voltage (or current)
`V(I)CONST`	Hold voltage (or current) constant
`V(I)UPDOWN`	Ramp voltage (or current) up and down
`V(I)STEP`	Double voltage (or current) steps
`V(I)MSTEP`	Multiple step voltage (or current) steps
`FRA`	Sine wave

Table 3. Available `CURVE` classes within Explain.

When `Signal` has been created it is assigned (or connected) to the `Pstat` object using the `Pstat.SetSignal` function. This is analogous to connecting a wire from the function generator to the analog potentiostat.

`CURVE`-class objects control data-acquisition, generation of plots in real time, and real-time stopping of data-acquisition functions (stop at current/voltage limits or stabilities). The following line of code

```
Curve = CPIV.New ("CURVE", Pstat)
```

creates a new object called `Curve`. It is an instance of the `CPIV` class. Table 4 contains a list of available `CURVE` classes within Explain.

Various different electrochemical experiments require different types of `CURVE` objects. Notice also that specific `SIGNAL` objects are used with each `CURVE` object. For example a `CPIV` requires a `Signal` that is an instance function of a `VRAMP` or `VCONST` class. These signal objects supply voltage information. `CPIV` would not function with a `Signal` created from the `IRAMP` class because this class supplies current.

`CURVE` objects also control the real-time plots during data acquisition. The following line of code sets the plot mode to linear current (*I*) versus voltage (*V*), which is one of the plot modes available for objects created from the `CPIV` class.

```
Curve.SetPlot (CPIV_LINIV, NIL, NIL, NIL)
```

The `Polarization Resistance.exp` script does not contain any run-time testing criteria in its `CPIV` object for stopping the data-acquisition. The addition of a maximum current limit to the `Polarization Resistance.exp` script is discussed in "Commonly Requested Changes to Explain™ Scripts," which is available by contacting Gamry or through the Gamry web site www.gamry.com/application-notes/.

A series of "bookkeeping" activities must follow the generation of the `CURVE` object before it can run the experiment. These include:

- Initializing the `Signal` object to ensure that it is in a known state:

```
Pstat.InitSignal ()
```

- Activating the cell switch on the potentiostat:

```
Pstat.SetCell (CellOn)  ; Turn on the
cell
```

- Displaying a message to the user in the lower left-hand corner of the window to inform if something is happening:

```
Notify ("Autoranging")
```

`CURVE` Class	**Description**
`CPIV`	Control potential and measure *I* and *V*
`CIIV`	Control *I* and measure *I* and *V*
`OCV`	Open-circuit voltage
`CV`	Cyclic voltammetry
`CGEN`	Curve GENeric
`NSCURVE`	NoiSe `CURVE`
`FRACURVE`	Electrochemical impedance spectroscopy `CURVE`
`IVT`	Measure *I* and *V* versus time
`GALVCOR`	GALVanic CORosion
`CCAL`	Calibration

Table 4. Some `CURVE` classes within Explain.

- Finding the best current-to-voltage converter range at which to start the experiment:

```
Pstat.FindIERange ()
```

- And again displaying a message that something is happening:

```
Notify ("Running Curve")
```

- Making the `Curve` object the active object allows the data to be displayed on the screen:

```
Curve.Activate ()
```

- And finally running the `Curve` object performs the experiment.

```
Curve.Run ()
```

- When the `Curve` object is finished running (the experiment is done), the potentiostat cell switch is turned off.

```
Pstat.SetCell (CellOff)
```

- And the data contained in the `Curve` object are written to the data file.

```
Curve.Printl ()
Dawdle ()
```

The program displays the active `Curve` object on the screen, waiting until you hit the `F2` key, because of the `Dawdle ()` function. When you hit the `F2` key, then the script returns to the point where the `CPIV` function was called. The output file is closed, the `Pstat` object is closed, and the experiment ends.

Conclusion

The Explain™ programming language provides an unrivaled degree of flexibility in customizing experiments while preserving a simple point-and-click interface. Standard experiments provided by Gamry can be performed with the click of a button. More-complicated experiments may be created from the scripts provided with the Gamry system by copying those scripts and making modifications. The Explain programming language and user-accessible object-oriented scripts provide an easily modified platform for electrochemical experiments.

Appendix A

Polarization Resistance.exp

```
; Explain Script for Polarization Resistance Experiment
; Copyright (c) Gamry Instruments, Inc.
; Version 6

include "explain6.exp"
include "DC105.exp"

function Main ()
      ; Create Objects which are used in the following Setup dialog
      PstatSelect = PSTATSELECT.New ("PSTAT", "&Pstat")

      Title   = LABEL.New ("TITLE", 80, "Polarization Resistance", "Test
&Identifier")
      Notes   = NOTES.New ("NOTES", 400, NIL, "&Notes...")
      Output  = OUTPUT.New ("OUTPUT", "POLRES.DTA","Output &File")

      Area    = QUANT.New ("AREA", 1.0, "Sample &Area (cm^2)")
      Density = QUANT.New ("DENSITY", 7.87, "Densit&y (g/cm^3)")
      Equiv   = QUANT.New ("EQUIV", 27.92, "Equiv. &Wt")
      BetaA   = QUANT.New ("BETAA", 0.120, "&Beta An.(V/Dec)")        ;1.05
      BetaC   = QUANT.New ("BETAC", 0.120, "&Beta Cat.(V/Dec)")   ;1.05

      VInit   = POTEN.New ("VINIT", -0.020, TRUE,"Initial &E (V)")
      VFinal  = POTEN.New ("VFINAL", 0.020, TRUE,"Final &E (V)")
      Scan    = QUANT.New ("SCANRATE", 0.125,"Scan Ra&te (mV/s)")
      Sample  = QUANT.New ("SAMPLETIME", 2.0,  "Sa&mple Period (s)")

      Condit  = TWOPARAM.New ("CONDIT", FALSE, 15.0, 0.0, "Conditionin&g", "Time(s)",
"E(V)")
      Delay   = TWOPARAM.New ("DELAY", FALSE, 300.0, 0.1, "Init. De&lay","Time(s)",
"Stab.(mV/s)")

      IRComp  = TOGGLE.New ("IRCOMP", FALSE, "IR Com&p")
      EQDelay  = QUANT.New ("EQDELAY", 0.0,      "Equil. &Time (s)")

      result = SetupRestore ("DC105.SET", "POLRES"
&                 ,PstatSelect.Selector (SELECTOR_ASTERISK)
&                 ,Title
&                 ,Output
&                 ,Notes
&                 ,VInit
&                 ,VFinal
&                 ,Scan
&                 ,Sample
&                 ,Area
&                 ,Density
&                 ,Equiv
&                 ,BetaA
&                 ,BetaC
&                 ,Condit
&                 ,Delay
&                 ,IRComp
&                 ,EQDelay
&                 )
```

```
	loop
		result = Setup ("Polarization Resistance"
&			,PstatSelect.Selector (SELECTOR_ASTERISK)
&			,Title
&			,Output
&			,Notes
&			,VInit
&			,VFinal
&			,Scan
&			,Sample
&			,Area
&			,Density
&			,Equiv
&			,BetaA
&			,BetaC
&			,Condit
&			,Delay
&			,IRComp
&			,EQDelay
&			)

		if (result eq FALSE)		; Cancel
			return

		;Pick the potentiostat
		Pstat = PstatSelect.CreatePstat ("PSTAT", "PstatClass")

		EstimatedPoints = CheckRampPoints (VFinal, VInit, Scan, Sample, 1000)
		OCStyle = CheckOCStyle (VInit, VFinal, VFinal)
		if (TestMaxPoints (EstimatedPoints, OCStyle))
			continue

		; Acquire output file
		if (Output.Open () ne TRUE)
			continue

		; Force Writing of File before Close
		Output.SetCommit (TRUE)

		if (CheckInstrument (Pstat, CALCHECK_DC) ne TRUE)
			continue

		; Acquire use of the requested Potentiostat.
		if (Pstat.Open () ne TRUE)
			continue

		break

	result = SetupSave ("DC105.SET", "POLRES"
&			,PstatSelect.Selector (SELECTOR_ASTERISK)
&			,Title
&			,Output
&			,Notes
&			,VInit
&			,VFinal
&			,Scan
&			,Sample
&			,Area
```

```
&                   ,Density
&                   ,Equiv
&                   ,BetaA
&                   ,BetaC
&                   ,Condit
&                   ,Delay
&                   ,IRComp
&                   ,EQDelay
&                   )

      Headline (Title.Value ())  ; Show user something's happening

      ; Write the settings for the experiment to the STDOUT window for later review

      Stdout ("Experimental Parameters")
      Stdout ("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^")
      Stdout ("Potentiostat: ", Config (GAMRYINI, Pstat.Section (), "Label"))
      Stdout ("Start Time:   ", TimeStamp ())
      Stdout ("")
      Stdout ("Initial Voltage:   ", VInit.Show ())
      Stdout ("Final Voltage:   ", VFinal.Show ())
      Stdout ("Scan Rate:   ", Scan.Value (), " mV/s")
      Stdout ("Sample Period:   ", Sample.Value (), " s")
      Stdout ("")
      Stdout ("Beta A:   ", BetaA.Value (), " V/Decade")
      Stdout ("Beta C:   ", BetaC.Value (), " V/Decade")
      Stdout ("")
      Stdout ("Specimen Area:   ", Area.Value (), " cm^2")
      Stdout ("Specimen Density:   ", Density.Value (), " g/cm^3")
      Stdout ("Specimen Equivalent Weight:   ", Equiv.Value (), " ")
      Stdout ("")
      if (Condit.Check ())
            Stdout ("Conditioning Time:   ", Condit.V1 (), " s @", Condit.V2 (), "
V")
      if (Delay.Check ())
            Stdout ("Delay Time:   ", Delay.V1 (), " s")
      if (IRComp.Value ())
            Stdout ("IR Correction:   On")
      else
            Stdout ("IR Correction:   Off")

      Notify2(Pstat.Label())

      ; Write Header info, setup data, etc. to output file.

      Printl ("EXPLAIN")
      Printl ("TAG\tPOLRES")
      Title.Printl ()
      Printl ("DATE\tLABEL\t", DateStamp (), "\tDate")
      Printl ("TIME\tLABEL\t", TimeStamp (), "\tTime")

      Notes.Printl ()

      Pstat.Printl ()
      VInit.Printl ()
      VFinal.Printl ()
      Scan.Printl ()
      Sample.Printl ()
```

```
	Area.Printl ()
	Density.Printl ()
	Equiv.Printl ()

	BetaA.Printl ()
	BetaC.Printl ()

	Condit.Printl ()
	Delay.Printl ()

	IRComp.Printl ()
	EQDelay.Printl ()

	; Condition the electrode
	if (Condit.Check ())
		if (Condition (Pstat, Condit.V1 (), Condit.V2 (), IRComp.Value
(),0.1*Area.Value ()) eq FALSE)
			return

	; Measure Eoc
	if (Delay.Check ())
		OCDelay (Pstat, Delay.V1 (), Delay.V2 () * 0.001)
	else
		OCDelay (Pstat, 10.0, NIL)
	Printl ("EOC\tQUANT\t", POTEN.Eoc (), "\tOpen Circuit (V)")

	;  Run the curve
	Cpiv (Pstat, VInit.VsEref ()
&		,VFinal.VsEref ()
&		,Scan.Value ()*0.001
&		,Sample.Value ()
&		,IRComp.Value ()
&		,EQDelay.Value ()
&		)

	Output.Close ()
	Pstat.Close ()

	return

; Run a controlled potential IV curve from VInit to VFinal
function Cpiv (Pstat, VInit, VFinal, ScanRate, SampleTime, IRToggle, EQDelay)

	InitializePstat (Pstat, IRToggle)

	Pstat.PrintHardwareSettings ()

	; Create a ramp generator for this pstat
	Signal = VRAMP.New ("SIGNAL", Pstat, VInit, VFinal, ScanRate, SampleTime)
	Signal.SetAcquisitionControl(NIL, NIL, NIL, SAMPLINGMODE_NR)

	Pstat.SetSignal (Signal)

	Curve = CPIV.New ("CURVE", Pstat)
	Curve.SetPlot (CPIV_LINIV, NIL, NIL, NIL)

	Pstat.InitSignal ()
```

```
    Pstat.SetCell (CellOn)  ; Turn on the cell

    Notify ("Autoranging")
    Pstat.FindIERange ()
    Pstat.SetIERangeLowerLimit (Pstat.IERange ())
    Equilibrate (Index (EQDelay))
    Notify ("Running Polarization Resistance Curve")
    Curve.Activate ()
    Curve.Run ()
    Pstat.SetCell (CellOff)
    Curve.Printl ()
    DisplayOverloadStatus(Curve)

    Notify ("Experiment done, press \"F2-Skip\" to continue")
    Dawdle ()

    return


function InitializePstat (Pstat, IRToggle)

    DC105InitializePstat (Pstat, IRToggle)

    Pstat.SetVchRange (2.0)

    return
```

PC6, ECM8, Gamry Framework, and Explain are trademarks of Gamry Instruments, Inc. Microsoft, Windows, and Visual Basic are registered trademarks of the Microsoft Corporation. Java is a trademark of Sun Microsystems, Inc.

Application Note *Rev. 2.0 3/20/2015 © Copyright* 1990–2015 Gamry Instruments, Inc.

734 Louis Drive • Warminster PA 18974 • Tel. 215 682 9330 Fax 215 682 9331 • www.gamry.com •info@gamry.com