

Customizing Electrochemical Experiments with the Explain™ Scripting Language

Introduction

Early electrochemical instruments were based on analog components that generated a waveform, controlled the cell, and recorded the data; for example, a function generator, a potentiostat, and an X-Y recorder. These systems allowed the user to generate almost any waveform (steps, ramps, sine waves) and record the data on chart paper. To perform more complicated or faster experiments, the user would interchange signal generators and recording devices to accommodate his needs. A negative side effect of this flexibility was that the instrument often consisted of three large boxes with interconnecting wires, not to mention the reams of paper used in recording experimental results.

The evolution of the desktop computer has significantly changed the world of electrochemical instrumentation. Computers and software have successfully replaced the function generator and X-Y recorder. In most cases, you use software written by the potentiostat manufacturer. The primary advantage of the computer-controlled instrument is automation and simplicity. These new instruments are smaller with fewer interconnecting cables. The data files are stored to disk and conveniently printed when needed. Procedures can often be controlled by the computer, allowing the researcher to focus on more important functions.

However, the computerization of electrochemical instruments has limited the flexibility to perform special experiments. The experimental sequence of events and control of the potentiostat were configured for predetermined experiments. If the scientist wants to change that order of events or change any parameter that is not in the setup screen, he must write his own experimental control program.

Gamry Instruments has developed an approach called Open Source Scripting to allow significant flexibility in computerized electrochemistry. The user-accessible Explain™ scripting language alleviates many of these problems on the data acquisition side of the experiment! You can easily make modifications to the standard experimental scripts that are provided with each Gamry software application package.

The following discussion introduces the major concepts 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. You should also review the 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. All of these notes are available for download from www.Gamry.com !

Data acquisition, of course, is only one part of the experiment. Data analysis is also a key function. 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. 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. Windows® Point-and-Click Environment

- You perform standard experiments such as cyclic voltammetry, electrochemical impedance spectroscopy, and polarization resistance by simply selecting the experiment from a pull down menu and filling in the appropriate setup parameters in the dialog boxes.

2. Explain™ Scripts

- Selecting an experiment from the pull-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 it in a graph on the screen, and saves the data to a file.

3. Compiled Code.

- The third layer of the Gamry Framework environment is the compiled code that implements the Gamry application and many of the objects that are used by Explain language.
- The user does not have 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 Windows dialog box called "Open a script for editing" should be visible (see Figure 1). The files listed in that window with the extension ".exp" are editable Explain scripts. Note that different Explain scripts will be available on your machine based upon the software application packages installed.

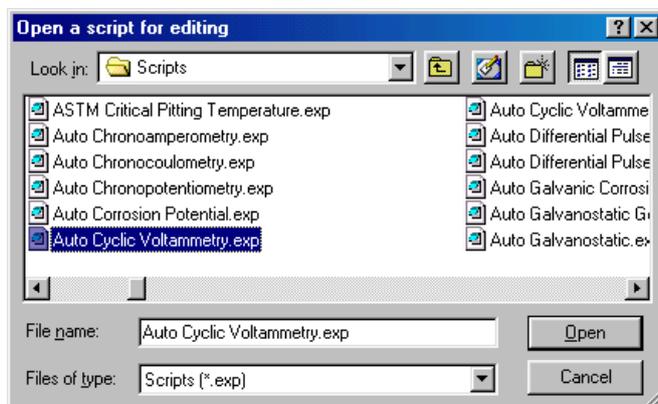


Figure 1. Selecting an Explain™ script for editing.

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 sidebar: Functions in the Explain™ Language.). When Explain encounters an unknown function it will automatically search 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 can be thought of as collections of physical items that share characteristics. Examples of classes are the different shapes, CIRCLE, SQUARE, and TRIANGLE (Note that by convention, classes are in uppercase in Explain). Each is a different class of objects. All objects that were a member of the CIRCLE class would know that they are round, an object that is a member of the SQUARE class would know 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 concept 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 will be created with the tag of My Circle, it will know that it is located at 2, 5 on the screen, it will have a radius of 10, and the circle will be 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()
```

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.

Notice that since this object was previously defined it knows the tag, coordinate, radius, and color information. Since 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 utilized 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, Inc., 1989-2001
; Version 4.0
```

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 will treat 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 functionality 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 "explain4.exp"
include "DC105.exp"
```

The `explain4.exp` script is contained in almost all other scripts since it contains a list of "includes" that add functionality to the Gamry Framework. Some of the more commonly included Explain scripts are listed in Table 1.

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

Explain Script	Function
Common4.exp Common Functions.exp	Library routines for functions and objects.
signal.exp curve.exp pstat.exp pc4.exp	Information regarding specific potentiostat hardware control
math.exp stats.exp	Mathematical routines.
Explain4.exp	Master include file – Includes all of the above files.
Muxlib.exp tdc1cal.exp tdc2cal.exp	Driver information for peripheral instruments, e.g., multiplexers and temperature controllers.

Table 1 Often used Explain "include" files

In the layout of an Explain script, the object definition section begins immediately following the function Main(). This will be 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 will display a setup window on the computer screen. This is done using the function Setup("Title", Item1, Item2, Item3, ...). The following code is used to display 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 ampersand symbol allows the Explain interpreter to ignore any indentations on these segments of code.

Once the user has entered the parameters for the experiment and hit the OK button, the setup window closes and the script moves on to create 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 has been closed.

If any error checking needs to be performed on the information entered in the setup box it is done following the closing of the setup box. In general, this will include 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 OK, the loop around that setup function is exited.

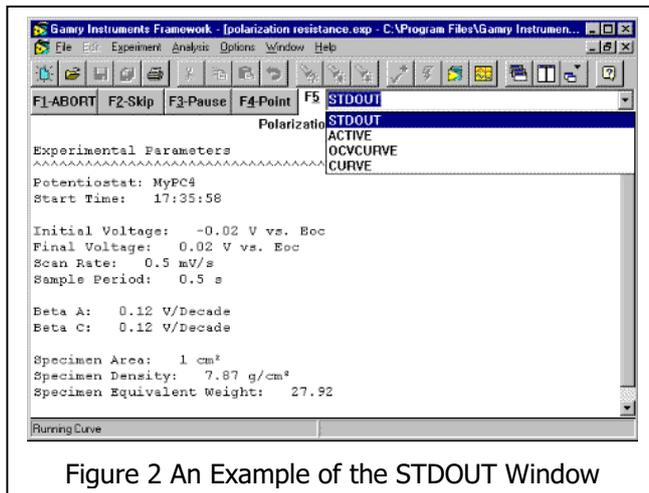


Figure 2 An Example of the STDOUT Window

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

Writing information to the STDOUT Window is performed using the function `Stdout()`. 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.

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.

Once the STDOUT and data file contain the setup parameters, the script moves on to control the potentiostat and perform the experiment. Before the actual polarization resistance scan starts, the script checks to see if the conditioning toggle box was 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` that is included in beginning of this script.


```

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

```

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\t", POTEN.Eoc ())

```

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 ()
&        )

```

Since this is a function call, the Gamry Framework will search the script to see where the function `Cpiv()` is defined and perform 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 are "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 `InitializePstat` 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 `InitializePstat` 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 discussed in the introduction. 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. The following is an abbreviated list of available `SIGNAL` classes. Additional `SIGNAL` classes may be added as we add 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

Once `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, real-time plot generation, 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. The following table contains a list of available `CURVE` classes within `Explain`.

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 vs. Time
GALVCOR	GALVanic CORosion
CCAL	Calibration

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 since this would supply 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) vs. 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.

A series of bookkeeping activities must follow the generation of the Curve object before it can be used to run the experiment. These include initializing the Signal object to ensure that it is in a known state:

```
Pstat.InitSignal ()
```

Turning on 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 let them know something is happening:

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

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

```
Pstat.FindIERange ()
```

And once again displaying a message that something is happening:

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

Making the Curve object the active object will allow the data to be displayed on the screen:

```
Curve.Activate ()
```

And finally running the Curve object performs the experiment.

```
Curve.Run ()
```

Once the Curve object is finished running (the experiment is over) the potentiostat cell switch is turned off:

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

And the data contained in the Curve object is written to the data file.

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

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

Sequencing Experiments

In some cases, the user may want to string several experiments together to form a complicated series of experiments. These sequences may also include time delays (i.e., wait for 30 minutes) or time triggers (i.e., wait until 10:00 a.m.).

In Explain, two special types of scripts are used to create sequences. These are called the "master" script and the "auto" script. A master script is an Explain script that contains a list of other scripts to be run and when to run them. Since the master script is written in Explain, you can create very flexible experiment sequences.

The master script calls the auto scripts that, in turn, run the experiments. The master script provides a way to schedule the timing and repetitions of the auto scripts. Appendix B is the `Runmany.exp`, which is an example of a master script that is delivered with the Gamry Framework software. This script consists primarily of a number of calls to the function `LaunchWait()`. The format for the `LaunchWait()` function is as follows:

```
if LaunchWait(AutoScriptFile, SetupFile, SetupName
&             , OutputFile, PstatNo, ChannelNo) eq FALSE)
return
```

Note that the use of an "if" statement before the `LaunchWait()` test requires that the experiment was successfully completed before moving on to the next experiment.

The *AutoScriptFile* must be an auto script, which will be discussed below. The *SetupFile* and *SetupName* are created by saving setup information using the save button in the setup window generated by the regular script. For example, the setup information for an automated polarization resistance experiment would be saved using the setup box from the `Polarization Resistance.exp` script. *OutputFile* is the name of the data file that will be created. *PstatNo* is the designation of the potentiostat in the computer that should be used to run the experiment (1 through 4 are available options, 1 being the default for a single potentiostat in a computer). *ChannelNo* is the channel on the ECM8™ multiplexer that will be used, a value of `NIL` is used when no multiplexer is present.

Auto scripts are experiments that must be launched by a master script. The major difference between an auto script and the corresponding normal script is that the auto script is designed to run with minimal operator interaction. Instead of asking the operator to supply experiment parameters, the parameters are read from setup files. You'll need to create setup files for each type of experiment to be performed. In addition, warning messages and queries requiring user input are avoided so the sequence does not pause waiting for an "OK".

Another difference is that the master script can pass parameters to the auto script. This allows parameter sets, file names, etc. to be controlled by the master script. A side effect is that you cannot call the auto scripts directly from the Gamry Framework since they expect input parameters.

You can create your own specialized auto scripts or use the auto scripts Gamry has added to the application packages. Both original and automated versions create the same data file formats so the analysis routines will work with data files from either.

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 provides an easily modified platform for electrochemical experiments.

© Copyright 2002 Gamry Instruments, Inc. All rights reserved.

PC4, 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 the trademark of Sun Microsystems, Inc.

Appendix A

Polarization Resistance.exp

```
; Explain Script for Polarization Resistance Experiment
; Copyright (c) Gamry Instruments, Inc., 1989-2001
; Version 4.0

include "explain4.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 (cm2)")
    Density = QUANT.New ("DENSITY", 7.87, "Densit&y (gm/cm3)")
    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")

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

```

&         ,Delay
&         ,IRComp
&     )

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

        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)

        ; 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

```



```

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

BetaA.Printl ()
BetaC.Printl ()

Condit.Printl ()
Delay.Printl ()

IRComp.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 ()
&     )

Output.Close ()
Pstat.Close ()

return

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

    InitializePstat (Pstat, IRToggle)

    PrintHardwareSettings (Pstat)

    ; Create a ramp generator for this pstat
    Signal = VRAMP.New ("SIGNAL", Pstat, VInit, VFinal, ScanRate, SampleTime)
    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 ())
Notify ("Running Curve")
Curve.Activate ()
Curve.Run ()
Pstat.SetCell (CellOff)
Curve.Print1 ()

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

return

function InitializePstat (Pstat, IRToggle)
    Pstat.SetCtrlMode (PstatMode)
    Pstat.SetCell (CellOff)
    Pstat.SetStability (StabilityNorm)
    Pstat.SetCASpeed (CASpeedMed)
    Pstat.SetConvention (Anodic)
    Pstat.SetGround (Float)
    Pstat.SetIchRange (3.0)
    Pstat.SetIchRangeMode (TRUE)
    Pstat.SetIchOffsetEnable (FALSE)
    Pstat.SetIchFilter (5.0)
    Pstat.SetVchRange (2.0)
    Pstat.SetVchRangeMode (TRUE)
    Pstat.SetVchOffsetEnable (FALSE)
    Pstat.SetVchFilter (5.0)
    Pstat.SetAchRange (3.0)
    Pstat.SetIERangeLowerLimit (NIL)
    Pstat.SetIERange (3.0E-5)
    Pstat.SetIERangeMode (TRUE)
    Pstat.SetAnalogOut (0.0)
    Pstat.SetVoltage (0.0)
    Pstat.SetPosFeedEnable (FALSE)
    Pstat.SetDummy (NormalCell)

    if (IRToggle)
        Pstat.SetIruptMode (IruptNorm, EuExtrap, IruptTime, POTEN.Eoc(), 1.0)
        Pstat.SetVchFilter (100000.0)
    else
        Pstat.SetIruptMode (IruptOff)

    return

```

Appendix B

Runmany.exp

```
; Explain Script for automatic scheduling of experiments.
; Copyright (c) Gamry Instruments, Inc., 1995-2001
; Version 4.0

; Use:  Runs a sequence of single experiments.

include "explain4.exp"
include "Auto Utilities.exp"

function Main()

    if (LaunchWait("Auto Corrosion Potential.exp"
&                , "auto.set"          ; Set File
&                , "CPOT"              ; Set Name
&                , "autocpot.dta"      ; Filename where data will be saved
&                , 1                    ; Potentiostat number
&                , NIL                  ; Mux Channel (NIL = No mux used)
&                ) eq FALSE)
        return

    if (LaunchWait("Auto RpEc Trend.exp"
&                , "auto.set"          ; Set File
&                , "RP"                ; Set Name
&                , "autorp.dta"        ; Filename where data will be saved
&                , 1                    ; Potentiostat number
&                , NIL                  ; Mux Channel (NIL = No mux used)
&                ) eq FALSE)
        return

    if (LaunchWait("Auto Potentiostatic EIS.exp"
&                , "auto.set"          ; Set File
&                , "EIS"               ; Set Name
&                , "autoeis.dta"       ; Filename where data will be saved
&                , 1                    ; Potentiostat number
&                , NIL                  ; Mux Channel (NIL = No mux used)
&                ) eq FALSE)
        return

    Notify("All Experiments Done")

    Dawdle()
```

